Arithmetic problems on the WORD model

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2012 ; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. All rights reserved.

Inputs and Input sizes

In most of the problems we had encountered before input size was a clearly discernible quantity. For polynomials it was the degree, for matrices the dimension of the matrix, for sorting the number of keys. We never bothered about the length of a key in sorting, nor whether the coefficients of a polynomial are integers, reals, or arbitrary precision numbers. We always assumed that all elementary arithmetic operations take 1 step to complete. In this section we are going to examine many of these same arithmetic operations we had ignored dealing with in the past. So such assumptions of constant time execution may be meaningless. Suppose we want to find the running time for integer multiplication; first we assume that the numbers are positive, as in any other case we can first establish the sign of the product and then multiply the absolute values of the two numbers, which are positive.

Problem 1: Integer Multiplication

Input: Two (positive) integers x and y.

Output: Product $z = x * y$.

It is obvious that problem size can NOT be 2, the number of inputs. The running time of multiplication depends on the number of bits or digits used to represent x and y. Therefore a more accurate description of the problem is

Problem 2: Integer Multiplication

Input: Two n-bit numbers x and y.

Output: Product $z = x * y$.

Problem size in the latter is obviously n the number of bits in x and y. Note that z needs at most $2n$ bits for its representation. The elementary school algorithm has running time $O(n^2)$. However if the input size is not explicitly given it can be derived by the largest of the two integers.

Problem 3: Integer Multiplication

Input: Two (positive) numbers x and y.

Output: Product $z = x * y$.

For the latter case if $x > y$ then $\lg x + 1$ bits are needed to represent x or y. If however $y > x$ then $\lg y + 1$ bits are needed. To avoid this case analysis we may also correctly claim that $\lg x + \lg y$ is the number of bits required to represent x or y. The running time of integer multiplication is then $O(\lg^2 x)$, $O(\lg^2 y)$, $O(\lg^2 x + \lg^2 y)$ respectively.

(c) Copyright A. Gerbessiotis. All rights reserved.

For the problem of sorting, the decision tree model was used to establish a worst-case running time lower bound of $\Omega(n \lg n)$ for comparison-based sorting algorithms. The assumption was that for sorting the one and only operation that matters is comparison of keys, and such a comparison takes unit time, and it doesnot matter for that if keys are integers or strings of length up to one billion characters.

For many arithmetic problems we are going to analyze them under two models: the `WORD` and `BIT` models.

1. `WORD` **model of computation.** It's the model under which, one arithmetic operation takes one step, i.e. the addition of two "numbers" (integer or real) takes ONE time step.

2. `BIT` **model of computation.** It's the model under which, one arithmetic operation takes a number of steps that is dependent on the bit operations required to complete it, i.e. the addition of two "numbers" (integer or real) requires a number of steps proportional to the length of the two numbers.

That is, if a and b are of length n and m respectively, $a + b$ takes time $O(max\{n, m\}) = O(n + m)$. Another way to state this is that $a + b$ takes time $O(\lg a + \lg b)$, where $\lg x$ is the base two logarithm of x. Integer a requires $O(\lg a)$ bits for its minimal representation.

Otherwise, the models have the same power that a programming language has, `for, while` loops, branching `if-then-else` statements, functions etc.

3. `Straight-line program (SLP)` **model of computation.** Sometimes we might restrict the model further by using what we call `the straight-line program` (SLP) model of computation. Such programs consist of a sequence of arithmetic operations, with no iterative controls (i.e. loops) nor branching. The operators are binary, and the operands can be either input parameters, constants (positive or negative) or intermediate results. A binary operation looks like a_i = b *op* c in the SLP model. Operation counts under SLP can follow the WORD or the BIT model; in most cases in the course, the former case will be followed.

(c) Copyright A. Gerbessiotis. All rights reserved.

Degrees and Degree bounds

A polynomial $A(x) = a_0 + \ldots + a_{n-1}x^{n-1}$ is of degree $n-1$ if $a_{n-1} \neq 0$. Then, any $k > n-1$ is the degree bound of $A(x)$. Thus a polynomial of degree bound n may have degree 0, 1, ... up to (inclusive) $n-1$. If two polynomial $A(x)$ and $B(x)$ of degree bounds a (i.e. of actual degree at most $a-1$) and b (i.e. of actual degree at most $b-1$) respectively are multiplied, the product $C(x) = A(x)B(x)$ is of actual degree $a+b-2$ i.e. its degree bound is $a+b-1$ which is at most $a+b$.

Although we will avoid the use of the term degree bound in the remainder, in any subsequent discussion a reference to a polynomial of degree n may also imply that its degree is less than n.

Coefficient Representation of a polynomial.
Given a polynomial $A(x)$ of degree bound n we represent it by the ordered tuple of its coefficients $(a_0, \ldots, a_{n-1})$.

Polynomial Evaluation
The operation that finds the value of a polynomial like $A(x)$ at a point $x = c$ is called polynomial evaluation and the simplest way to evaluate $A(c)$ is by using Horner's rule as it was discussed in Subject 3.

Therefore for polynomial evaluation give $A(x)$ in the form $(a_0, \ldots, a_{n-1})$, and a number c we wish to find $A(c)$.

Point-value Representation of a polynomial.
A polynomial $A(x)$ of degree $n-1$ can also be uniquely represented by n different point-value pairs, i.e. the tuple

$$((x_0, A(x_0)), \ldots, (x_{n-1}, A(x_{n-1}))) \text{ where } x_i \neq x_j \text{ for all } i, j$$

Given these n pairs the coefficients of $A(x)$ can easily be found by an operation that is known as interpolation using Lagrange's formula in $O(n^2)$ time.

(c) Copyright A. Gerbessiotis. All rights reserved.

Theorem. The $(a_0, \ldots, a_{n-1})$ representation is equivalent to the point-value representation.

$$((x_0, A(x_0)), \ldots, (x_{n-1}, A(x_{n-1}))) \text{ where } x_i \neq x_j \text{ for all } i, j$$

Proof.

If we are given the vector representation of a polynomial the point-value can be easily obtained in $O(n^2)$ time.

However, it is not obvious if we are given the point-value representation how to obtain the cofficient of a polynomial. A theorem/formula due to **Lagrange** is the solution.

Lagrange's formula Consider n distinct point-value pairs $((x_1, b_1), \ldots, (x_n, b_n))$. We want to find the $n-1$ degree polynomial $A(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ such that $A(x_i) = b_i$, $1 \leq i \leq n$.

$$g_i(x) = \frac{(x - x_1)(x - x_2)\ldots(x - x_{i-1})(x - x_{i+1})\ldots(x - x_n)}{(x_i - x_1)(x_i - x_2)\ldots(x_i - x_{i-1})(x_i - x_{i+1})\ldots(x_i - x_n)}$$

It is easy to verify that $g_i(x_i) = 1$ as the numerator and denominator cancel out, and $g_i(x_j) = 0$ for $j \neq i$ as $(x_j - x_j)$ appears in the numerator and the corresponding term in the denominator is $x_i - x_j$.

If we consider the polynomial derived from all $g_i(x)$ such that

$$g(x) = b_1 g_1(x) + \ldots + b_n g_n(x)$$

we observe that $g(x_i) = b_i$ for all $i = 1, \ldots, n$.

Therefore

$$A(x) = g(x).$$

(c) Copyright A. Gerbessiotis. All rights reserved.

How to share a secret secretly

There is a group of N people who want to share a secret s in such a way that the presence of at least n of the N people would be required to recover s.

Method: Split secret s into N pieces $s_1, \ldots, s_N$ so that a quorum of n pieces is needed to retrieve s.

Justification: Redundancy. Think of N disk-drives and of a record s of a file. Even if $N - n$ disk-drives fail redundancy allows the recovery of s.

Theorem Shamir. There exists a very simple secret sharing scheme of low computational complexity.

Proof. Choose randomly a polynomial of degree $n - 1$. Let $f(x)$ be such a polynomial.

$$f(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + s$$

Choosing a polynomial randomly is equivalent to choosing its $n - 1$ coefficients $a_1, \ldots, a_{n-1}$ randomly. The 0-th degree coefficient will be our secret s.

The i-th piece of s is $s_i = f(i)$, $1 \leq i \leq N$. Piece s_i is assigned to person i.

Proposition 1. Polynomial $f(x)$ satisfies the requirements of secret sharing.

Suppose a quorum of n is gathered. Let these be $j_1, j_2, \ldots, j_n$. Then we have n point-value pairs $(j_i, f(j_i) = s_{j_i})$ of polynomial $f(x)$ and therefore by Lagrange's formula we derive that $f(x) = F(x)$, where

$$F(x) = b_1 g_1(x) + \ldots + b_n g_n(x)$$

In order to determine the secret s we evaluate $F(x)$ as found by Lagrange's formula at $x = 0$. Then, $s = F(0) = f(0)$.

Note. In practice the polynomial $f(x)$ is constructed over a finite field; very simply, think of s, N, and all important parameters to (eg. a_i) to be values up to say a prime number P and all computations are mod$P$.

(c) Copyright A. Gerbessiotis. All rights reserved.

A brute-force solution

Suppose we are given a polynomial $f(x)$ such that

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$$

where the indeterminate is x and the coefficients $a_i \in R$. Depending on the circumstances a_i could belong to the set of integer numbers (denoted by Z) or the set of rational numbers (denoted by Q). Suppose we are given $c \in R$. Polynomial evaluation deals with finding

$$f(c).$$

In order to compute $f(c)$ the naive method for evaluation first finds

$$1, c, c^2, \ldots, c^n \text{ for a total of } (n-1) \text{ multiplications}$$

then computes all products of c^i with a_i

$$a_0, a_1 c, a_2 c^2, \ldots, a_n c^n \text{ for a total of } n \text{ multiplications}$$

and finally adds up the partial products to derive $f(c)$

$$f(c) = a_0 + a_1 c + a_2 c^2 + \ldots + a_n c^n \text{ for a total of } n \text{ additions}$$

```
   Naive(a[0..n],n,x)
1. result = a[0]; xn=1;
2. for(j=1;j<=n;j++) {
3.  xn *= x;
4.  result = result + xn * a[j];
5. }
6. return(result);
```

(c) Copyright A. Gerbessiotis. All rights reserved.

Therefore the total cost of this method is : $(2n-1)$ M's (for multiplications) and n A's (for additions).

There is a faster way to perform polynomial evaluation however using a method that is known as Horner's method/rule or Horner-Newton rule. According to this method

$$f(x) = ((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0.$$

i.e. $f(x) = a_3x^3 + a_2x^2 + a_1x + a_0 = ((a_3x + a_2)x + a_1)x + a_0$. This method requires n M's and n A's, i.e. it is faster than the naive method.

```
   Horner(a[0..n],n,x)
1. result = a[n];
2. for(j=n-1;j>=0;j--) {
3.  result = result * x + a[j];
4. }
5. return(result);
```

Special Case: Exponentiation x^n

Question How fast can we raise x to the power of n, i.e. computer x^n for some input number x and integer n?

Theorem Exp x^n can be evaluated in $2\lg n$ M's.

Proof. Let $k = \lfloor \lg n \rfloor$. We first compute all power of two products

$$x, y_1 = x^2, y_2 = y_1^2 = x^4, y_3 = y_2^2 = x^8, \ldots y_k = y_{k-1}^2 = x^{2^k}$$

This requires k multiplications (`Count1`) by performing repeated doubling. Note that $y_k \leq x^n \leq y_k^2$.

We then find the binary representation of n i.e. $n = b_0 + b_1 2^1 + \ldots + b_k 2^k$, where $b_i = 0, 1$. Then

$$x^n = x^{b_0 + b_1 2^1 + \ldots + b_k 2^k} = x^{b_0} x^{b_1 2^1} \ldots x^{b_k 2^k}$$

In general, let m of the b_i's be 1 (i.e. the remaining b_i's are zero), where $m \leq k+1$. Let these m b_i's be $b_{l_1}, b_{l_2}, \ldots b_{l_m}$. Then $x_n = x^{2^{l_1}} x^{2^{l_2}} \ldots x^{2^{l_m}} = y_{l_1} y_{l_2} \ldots y_{l_m}$, i.e. x_n requires $m - 1 \leq k + 1 - 1 = k$ multiplications (`Count2`) for a total of $k + k = 2k = 2\lfloor \lg n \rfloor \leq 2 \lg n$ M's, if we combine `Count1` and `Count2`.

```
   Power( x, n)
1.  k = floor(lgn); X[0]=x;
2.  for (i=1;i<=k;i++)
3.   X[i] = X[i-1]*X[i-1] // X[i]=x**(2^i)
4.  n= B[k]B[k-1]...B[0] //binary representation of n
5.  if (B[0] == 1)  P=X;
6.   else           P=1;
7.  for(i=1;i<=k;i++);
8.   if (B[i]==1)  P= P* X[i];
9.  return(P);
```

Example. Let $n = 13$. Compute x^n for some x. Since $n = 13$, then $k = 3$. We first compute, $y_1 = x^2$, $y_2 = y_1^2 = x^4$, and $y_3 = y_2^2 = x^8$ with three M's. Given that the binary representation of 13 is 1101, then $x^{13} = x^{1\cdot 8 + 1\cdot 4 + 0\cdot 2 + 1\cdot 1} = y_3 \cdot y_2 \cdot x$, i.e. x^n can be computed with 2 additional M's for a total of 5 multiplications.

(c) Copyright A. Gerbessiotis. All rights reserved.

A short excursion to induction.

Statement. Let P_n be a statement (eg. proposition) that depends on integer variable n. Suppose we want to prove P_n is true by induction. We follow the steps below.

Base Case We show directly that for some small constant value of n, say n_0 that P_n is true for $n = n_0$ i.e. that P_{n_0} is true. This becomes the `base case` of the inductive proof. When induction is complete (and proved correctly and completely) what we will have proved is that P_n is true for all $n \geq n_0$.

In most cases, the base case is P_0 or P_1.

Inductive hypothesis. We formulate the `inductive hypothesis` that says that
"if P_{n-1} is true then P_n will be true".
It doesn't matter whether we have $n-1$ on the left side and n on the right or n on the left and $n+1$ on the right. However choosing one or the latter might make the proof easier or more difficult in some cases.

In order to complete the inductive proof, we prove the inductive hypothesis. The step in the proof that shows that the inductive hypothesis is valid is called the `inductive step`.

Inductive step.

Note. This is a very subtle step in an inductive proof; If one is not careful, the one can prove by induction that "all cows are brown".

We assume P_{n-1} is valid, and try to deduce the validity of P_n.

Why does induction work? If we show in the base case that P_0 is true, and have shown the inductive hypothesis to be true in the inductive step, then by applying the inductive hypothesis for $n = 1$ (i.e. the fact that if P_0 is true, then P_1 will be true) we show that P_1 will be true. Do the same again for P_1, and we show that P_2 is true, and continue ad infinitum.

A Lower Bound on Additions

Lemma 1. Any straight line program to compute $a_0 + \ldots + a_n$ requires at least n A/S's (additions or subtractions).

What is a straight line program? A straight line program is a program that consists of statements of the form $x_i = x_j + x_k$, where x_j, x_k are constant or variables and x_i is a variable. For example

```
t2 = a3 * c;
t3 = t2 + a2;
t4 = t3 * c;
t5 = t3 + a1;
t6 = t5 * c;
t7 = t6 + a0; computes a0 + a1 c + a2 c**2 + a3 c**3 in t7.
```

Proof. Proof by induction on n. Let P_n be the proposition that $a_0 + \ldots + a_n$ requires at least n A/S's. Let S_n be any straight line program that computes $a_0 + \ldots + a_n$ and would thus require if P_n was true at least n A/S's.

Base Case. For $n = 0$ the result is obvious since P_0 is true by default.

Inductive hypothesis. The inductive hypothesis is "if P_{n-1} is true, the P_n will be true" or equivalently "if S_{n-1} requires at least $n - 1$ A/S's, then S_n will require at least n A/S's.

(c) Copyright A. Gerbessiotis. All rights reserved.

Inductive Step. Let us the consider the evaluation of $a_0 + \ldots + a_n$ and let S_n be any straight line program for computing it. Then if in program S_n we substitute 0 for a_n we could get a SLP $S' = S_n(a_n = 0)$ that evaluates $a_0 + \ldots + a_{n-1}$. Thus S' is an S_{n-1} program. We will show that if S' requires at least $n-1$ A/S's P_n requires at least n A/S's.

Let $a = b$ op c be the first $\pm$ operation of S_n that uses a_n. We call this operation O. In such an operation one of b or c may be a_n or one of them is a multiple of a_n computed earlier; neither b nor c can be of any of the forms $b = \pm a_n \pm d$ or $c = \pm a_n \pm e$, since then not O but that operation would be the earliest one using a_n.

O MUST EXIST somewhere otherwise the final result is a multiple of a_n (which in general is absurd) as all operations that use a_n should be $*$ operations. This can not be true in general for arbitrary choices of a_0 through a_n.

We then do the following:

A1. In all operations preceding O we substitute 0 for a_n.

A2. In O we substitute 0 for b or c depending on whether b or c is the operand that is a_n or uses a_n.

A3. In all operations following O we subsitute 0 for a_n.

The cases of interest for O in A2 (be reminded that we are interested in a $\pm$ lower bound) are

1. $a = b + 0$.
2. $a = b - 0$.
3. $a = 0 + c$.
4. $a = 0 - c$.

(c) Copyright A. Gerbessiotis. All rights reserved.

proof of lower bound continued

In each one of cases 1-3 we eliminate O and replace a in the remainder by b (cases 1 and 2) or c (case 3) as required.

For case 4 we replace the step $a = 0 - c$ that involves a subtraction by the step $a = (-1) * c$, that involves a multiplication of c with constant -1.

In all four cases we reduced A/S's by one (by either eliminating an A/S or replacing in case 4 a subtraction with a multiplication).

The resulting program $S_{n-1} = S'$ is a program that clearly computes $a_0 + \ldots + a_{n-1}$. If $S_{n-1} = S'$ requires at least $n-1$ A/S's, then S_n that generated S' by eliminating a single A/S operation (and substituting 0 for a_n) would require at least n A/S's. This completes the induction. □

Lemma 2. Horner's rule requires at least n A/S's.

Proof of Lemma 2. Consider any straight-line program P that implements Horner's rule. Let it perform N Additions/Subtractions. Consider the case of evaluating $f(1)$ i.e. the special case where $x = 1$. In P it suffices to replace all occurrences of x by 1 (or equivalently rename x by 1) to compute $f(1)$, which incidentally is $a_0 + \ldots + a_n$. The number of $\pm$ steps in this renamed version of P remains the same as in the original P. By Lemma 1 the renamed version of P must have at least n additions/subtractions, i.e. $N \geq n$. Since the program is still P (other than the renaming of x by 1), we conclude that any program like P that does Horner's rule must have $N \geq n$ Additions/Subtractions (since otherwise we could have derived a program for evaluating $a_0 + \ldots + a_n$ in fewer than n A/S's).

This concludes the lower bound proof for polynomial evaluation.

Conclusion. Horner's method is optimal as far as additions and subtractions are concerned.

Is it optimal in terms of multiplications? No, if for example preprocessing is allowed.

(c) Copyright A. Gerbessiotis. All rights reserved.

The standard method for multiplying polynomials, multiplies two polynomials of degree $n-1$ (or degree bound n) in $O(n^2)$ time. We will show how to reduce this to $O(n^{\lg 3})$ by the end of this subject, and later on, how by using FFT, and combining ideas from the discussion below it can be further reduced to $O(n \lg n)$.

Another way to multiply polynomials is through Interpolation.

Point-value Representation of the product of two polynomials.

Given $A(x)$, $B(x)$ the product $C(x) = A(x)B(x)$ is of degree $2n-2$. Therefore its point-value representation requires $2n-1$ pairs to be known in $2n-1$ distinct values. The point-value representation of $A(x)$ and $B(x)$ are available, however, in only n points as they are both of degree $n-1$. Therefore we need to represent $A(x)$ and $B(x)$ in an extended form of $2n-1$ point-value pairs for this representation to be useful in establishing $C(x)$'s point-value pair representation.

Then let $((x_0, a_0), (x_1, a_1), \ldots, (x_{2n-1}, a_{2n-1})$ be the point-value representation of $A(x)$ in extended form. Let similarly $((x_0, b_0), (x_1, b_1), \ldots, (x_{2n-1}, b_{2n-1})$ be the point-value representation of $B(x)$ in extended form. Then since $C(x) = A(x)B(x)$ we know that $C(x_i) = A(x_i)B(x_i)$.

Therefore the point-value representation of $C(x)$ is $((x_0, a_0b_0), (x_1, a_1b_1), \ldots, (x_{2n-1}, a_{2n-1}b_{2n-1})$, and can be obtained from those of $A(x)$, $B(x)$ by performing $2n-1$ multiplications only i.e. it's very efficient compared to performing a polynomial multiplication of $A(x)$ and $B(x)$ to obtain $C(x)$ and then evaluating $C(x)$ at $2n-1$ or $2n$ points for a total cost of $\Theta(n^2)$.

Given this representation, $C(x)$ can be derived very easily by an additional interpolation operation.

(c) Copyright A. Gerbessiotis. All rights reserved.

Therefore there are two ways to find $C(x) = A(x)B(x)$ given $A(x), B(x)$.

```
   NaiveMultiply(A(x),B(x))
1. C(x)=A(x)* B(x);
2. return(C(x));
```

```
    IntelligentMultiply(A(x),B(x))
*1. Evaluate A(x) at 2n points x_i.
*2. Evaluate B(x) at 2n points x_i.
 3. Pairwise multiply A(x_i)B(x_i)
 4. Since C(x_i) = A(x_i) B(x_i)
    Interpolate to find C(x)
 5. return(C(x));
```

```
* : We only need evaluate the polynomials at 2n-1 points. The redundancy
    (one extra evaluation is to simplify exposition).
```

In the naive method step 1 requires $O(n^2)$ time.

In the intelligent method step 3 requires $O(n)$ time. If steps 1,2, and 4 are performed naively they would all require $O(n^2)$ time and this method becomes dumber than the naive method. If we choose, however, the x_i intelligently then, steps 1 and 2 can be performed in $O(n \lg n)$ time using a divide-and-conquer method, and step 4 becomes an operation similar to those of steps 1 and 2 and can also be performed in time $O(n \lg n)$. Thereforee total running time for the Intelligent method becomes $O(n \lg n)$ compared to the $O(n^2)$ of the naive approach.

As a final note, whether $A(x)$ is evaluated at $2n$ points or $2n-1$ is not that important. It helps the fact that $2n$ is an even number, and so some extra redundancy makes calculations easier.

(c) Copyright A. Gerbessiotis. All rights reserved.

Complex Numbers

A complex number z is represented by $z = a+ib$, where a, b are real numbers and symbol i is such that $i^2 = -1$. Sometimes we represent $a + ib$ by the pair (a, b).

For two complex numbers $z_1 = a + ib$ and $z_2 = c + id$, $z_1 + z_2 = (a + c) + i(b + d)$.

The product of z_1 and z_2 is $z_1 z_2 = (ac - bd) + i(ad + bc)$.

Total number of operations to multiply z_1 and z_2 are:

Additions/Subtractions : 2.

Multiplications : 4 (ac, bd, ad, bc).

If you don't want to remember this rule, do the following.

Write $z_1 = a + xb$ and $z_2 = c + xd$, where $x = i$.

Multiply $z_1 z_2$ as if you multiply two polynomials: $z_1 z_2 = ac + (bc + ad)x + bdx^2$.

Then recall that $x = i$ and thus $x^2 = i^2 = -1$, and $x^3 = -i$ and so on. $z_1 z_2 = ac+(bc+ad)i+bdi^2 = (ac-bd)+i(ad+bc)$.

The product of z_1 and z_2 is $z_1 z_2 = (ac - bd) + i(ad + bc)$ again.

So complex number multiplication and polynomial multiplication are "equivalent".

Question 1. Can we do better? Can you multiply $z_1 z_2$ in fewer than 4 (say 3) multiplications? (We don't care how many Additions/Subtractions will be required). Show How.

Question 2. Can you do it in 2 multiplications?

Question 3. How can you use the solution of Question 1, to multiply polynomials faster that $O(n^2)$?

Hint. Let $Z_1(x) = A(x) + I(x)B(x)$ and $Z_2(x) = C(x) + I(x)D(x)$, i.e. decompose/rewrite the polynomials Z_1 and Z_2. Suppose that $I(x)$ is an easy to deal with polynomial (i.e. $I(x) = x$ or $I(x) = x^k$ for some wisely chosen k). How many multiplications can you use to multiply the two polynomials Z_1 and Z_2?

(c) Copyright A. Gerbessiotis. All rights reserved.