Problem 1. (30 points)

Give a divide-and-conquer algorithm for multiplying two polynomials of degree $n$ in time $O(n \log_2 3)$.

**Answer:** Suppose the two polynomials we want to multiply are $A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$ and $B(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_{n-1} x^{n-1}$. We assume that $n$ is a power of two (otherwise, we can always pad the coefficients with zeros to reach the "next" power of two). Let us break $A(x)$ and $B(x)$ into two polynomials as follows.

$$
\begin{align*}
A(x) & = A_0(x) + x^{n/2} A_1(x) \\
B(x) & = B_0(x) + x^{n/2} B_1(x)
\end{align*}
$$

where

$$
\begin{align*}
A_0(x) & = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n/2-1} x^{n/2-1} \\
A_1(x) & = a_{n/2} + a_{n/2+1} x + a_{n/2+2} x^2 + \ldots + a_{n-1} x^{n/2-1} \\
B_0(x) & = b_0 + b_1 x + b_2 x^2 + \ldots + b_{n/2-1} x^{n/2-1} \\
B_1(x) & = b_{n/2} + b_{n/2+1} x + b_{n/2+2} x^2 + \ldots + b_{n-1} x^{n/2-1}
\end{align*}
$$

Then the problem of multiplying $A(x)B(x)$ can be decomposed in the problem of multiplying $A_0(x), A_1(x), B_1(x), B_1(x)$ as follows. We omit "($x$)" to reduce the clutter.

$$
\begin{align*}
AB & = (A_0 + x^{n/2} A_1)(B_0 + x^{n/2} B_1) \\
& = A_0 B_0 + x^{n/2} (A_0 B_1 + A_1 B_0) + x^n A_1 B_1 \\
& = A_0 B_0 + x^{n/2} ((A_0 - A_1)(B_1 - B_0) + A_0 B_0 + A_1 B_1) + x^n A_1 B_1
\end{align*}
$$

Therefore, we need 3 multiplications of two polynomials of degree $n/2$ (namely, $A_0 B_0$, $A_1 B_1$ and $(A_0 - A_1)(B_1 - B_0)$) and $O(n)$ additional work for the sum and the differences.

The recurrence relations is

$$
T(n) = \begin{cases} 
\Theta(1) & n = 1 \\
3T(n/2) + O(n) & n > 1 
\end{cases}
$$

which can be solved using the Master Theorem, concluding that $T(n) \in O(n \log_2 3)$.

Problem 2. (40 points)

The *square* of a matrix $A$ is its product with itself, $AA$.

- Show that five multiplications are sufficient to compute the square of a $2 \times 2$ matrix.
- What is wrong with the following algorithm for computing the square of a $n \times n$ matrix?
“Use a divide-and-conquer approach as in Strassen’s algorithm, except that instead of getting 7 subproblems of size $n/2$ we now get 5 subproblems of size $n/2$ thanks to the previous observation. Using the same analysis as in Strassen’s algorithm, we can conclude that the new algorithm runs in time $O(n^{\log_2 5})$.”

**Answer:** Assume that $A$ is a $2 \times 2$ matrix, that is

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix},$$

then

$$A^2 = \begin{bmatrix} a^2 + bc & ab + bd \\ ac + cd & bc + d^2 \end{bmatrix} = \begin{bmatrix} a^2 + bc & b(a + d) \\ c(a + d) & d^2 + bc \end{bmatrix}$$

Only 5 multiplications are necessary, namely $a^2$, $d^2$, $b(a + d)$, $bc$, and $c(a + d)$. However, we cannot achieve the same result for a $n \times n$ matrix. If

$$A = \begin{bmatrix} P & Q \\ R & S \end{bmatrix},$$

then

$$A^2 = \begin{bmatrix} P^2 + QR & PQ + QS \\ RP + SR & RQ + S^2 \end{bmatrix}$$

but we cannot achieve five multiplication as for the case of the scalars, because matrix multiplication is not commutative. Also, note that not all these subproblems involve squaring, most of them are regular matrix multiplications. So the nature of the original problem $A^2$ is different from the nature of the subproblems (a mix of squaring and multiplications).

In fact, squaring matrices is no easier than matrix multiplication. It is not too hard to show that if $n \times n$ matrices can be squared in time $O(n^c)$, then any two $n \times n$ matrices can be multiplied in time $O(n^c)$.

**Problem 3. (30 points)**

Given an array $A$ of $n$ (possibly negative) integers, find two indices $1 \leq i \leq n$ and $1 \leq j \leq n$ such that the value of $\sum_{k=i}^{j} a_k$ is maximized.

Write an $O(n \log n)$-time divide and conquer algorithm for the problem described above. The algorithm should return $i$ and $j$. If all elements of the array are negative, the algorithm should return $i = j = 0$.

**Answer:** The general strategy is based on divide and conquer.

1A $O(n)$ dynamic programming algorithm for this problem exists, but here you are supposed to give the slower divide and conquer algorithm.
Divide the array in two halves. Either the maximum is on the left half, or the maximum is on the right half, or the maximum is the sum of a region on the left half all the way to the center and a region on right half starting from the center.

Algorithm maxsub(p, q)
    if p + 1 = q then { if A[p] < 0 then return 0 else return A[p] }
    else { /* p + 2 ≤ q */
        k ← ⌊(p + q)/2⌋
        lmax ← maxsub(p, k)
        rmax ← maxsub(k + 1, q)
        lbdmax ← 0
        lbd ← 0
        for i ← k to p
            lbd ← lbd + a[i]
            if lbdmax < lbd then lbdmax ← lbd
        rbdmax ← 0
        rbd ← 0
        for i ← k + 1 to q
            rbd ← rbd + a[i]
            if rbdmax < rbd then rbdmax ← rbd
        return max(lmax, rmax, lbdmax + rbdmax)
    }

We solve the problem by calling the function maxsub(1, n).
Time complexity: This algorithm has the same recurrence relation of Mergesort, which has solution \(O(n \log n)\).