# Homework 1: Divide and Conquer
# CS 6161: Design and Analysis of Algorithm (Fall'20)

Due: 6 pm, September 16 (Wednesday)

**General Instructions** The assignment is meant to be challenging. Feel free to discuss with fellow students, however please write up your solutions independently (e.g., start writing solutions after a few hours of any discussion) and acknowledge everyone you discussed the homework with on your writeup. The course materials are all on UVA Collab. You may refer to any materials covered in our class. However, any attempt to consult outside sources, on the Internet or otherwise, for solutions to any of these homework problems is *not* allowed.

Whenever a question asks you to design or construct an algorithm, please formally write the pseudo-code in an algorithm box with all the essential elements (the input, output, and operations); whenever a question asks you to "show" or "prove" a claim, please provide a formal mathematical proof. Unless specified otherwise, in every problem set, when we ask for an answer in asymptotic notation, we are asking either for a $\Theta(\cdot)$ result, or else the tightest $O(\cdot)$ result you can come up with.

**Submission Instructions** Please write your solutions in LaTeXusing the **provided template**, hand-written solutions will not be accepted. **You must submit your answer in PDF version via Gradescope.** You should have received a notice about signing up for Gradescope — if not, please let the TAs know as soon as possible. Hope you enjoy the homework!

## Problem 1

1. **[8 Points]** Review the definition of asymptotic notions. Now given a set of functions, $S = \{h, k, m\}$ where $h(n) = \sqrt{n} + \log n$, $k(n) = 2^n/n^9$, $m(n) = (\log n)^9 + n^{1/2}$.

   For any pair of functions $f, g$ from the above set of functions, say whether it holds that $f = O(g)$, $f = \Omega(g)$, $f = o(g)$, $f = \omega(g)$ or $f = \Theta(g)$. If multiple relations between $f$ and $g$ hold, state the stronger one. For example if $f = o(g)$ and also $f = O(g)$, then only say $f = o(g)$, or alternatively $g = \omega(f)$. **Note:** *Give a short argument for each of your answers in this problem.*

**Solution:**

$$h(n) = o(k(n)) \iff k(n) = \omega(h(n))$$
$$m(n) = o(k(n)) \iff k(n) = \omega(m(n))$$
$$h(n) = \Theta(m(n)) \iff m(n) = \Theta(h(n))$$

2. **[10 Points]** Consider the following functions, and rearrange them in ascending order of growth rate. That is, if function $g(n)$ comes after function $f(n)$ in your list, then it should be the case that $f(n) = O(g(n))$.

$$\begin{aligned} f_1(n) &= n^{\sqrt{\log n}} \\ f_2(n) &= 7n + o(n) \\ f_3(n) &= 100\sqrt{n} + 2n^{1/4} + 100000 \\ f_4(n) &= 2^{\log^2 n} \\ f_5(n) &= 2^{2\log n} + n^{1.5} \\ f_6(n) &= 3^n \\ f_7(n) &= \ln(n!) \\ f_8(n) &= 2^{1.5^n} \end{aligned}$$

**Note:** *Be careful about the superscript in the mathematical expressions*

**Solution:**

$$\begin{aligned} f_1(n) &= n^{\sqrt{\log n}} = \omega(n^2) = \Theta(2^{\log(n)\sqrt{\log n}}) = \Theta(2^{\log^{1.5} n}) \\ f_2(n) &= 7n + o(n) = \Theta(n) \\ f_3(n) &= 100\sqrt{n} + 2n^{1/4} + 100000 = \Theta(\sqrt{n}) \\ f_4(n) &= 2^{\log^2 n} = \Theta(2^{\log^2 n}) \\ f_5(n) &= 2^{2\log n} + n^{1.5} = \Theta(n^2) \\ f_6(n) &= 3^n = \Theta(3^n) \\ f_7(n) &= \ln(n!) = \Theta(n\log(n)) \\ f_8(n) &= 2^{1.5^n} = \omega(2^{1.5n}) = \omega(3^n) \end{aligned}$$

$$f_3(n), f_2(n), f_7(n), f_5(n), f_1(n), f_4(n), f_6(n), f_8(n)$$

3. **[12 Points]** Assume you have functions $f$ and $g$ such that $f(n) = O(g(n))$. For each of the following statement, decide whether you think it is true of false and give a proof or counterexample. You may assume that all functions mentioned are monotone non-decreasing and non-negative.

   (a) $\log_2 f(n) = O(\log_2 g(n))$

   (b) $2^{f(n)} = O(2^{g(n)})$

   (c) $f(n)^2 = O(g(n)^2)$

**Solution:** (a) False, $f(n) = 2, g(n) = 1$

(b) False, $f(n) = 2n, g(n) = n$

(c) True

## Problem 2

**[10 Points]** Consider the recurrence relation $T(n) = T(n-10) + n$ for $n > 10$, with $T(n) = 1$ for all $n \leq 10$. Your friend claims that $T(n) = O(n)$, and offers the following justification:

Let's use the Master Theorem with $a = 1, b = \frac{n}{n-10}$, and $d = 1$. This applies since, for any $n > 10$, we have

$$\frac{n}{b} = n \cdot \left(\frac{n-10}{n}\right) = n - 10$$

Then for any $n > 10$, we have $a < b^d$, so the Master Theorem says that $T(n) = O\left(n^d\right) = O(n)$

Do you agree with your friend's argument? If not, what is the correct answer? If it helps, you may assume that $n$ is a multiple of 10.

**Note:** *To convince your friend, you need to clearly identify the faulty logic above, then give your solution to this recurrence along with a short but convincing justification (no formal proof needed).*

**Solution:** Do not agree. The Master Theorem only applies when $a, b$ are constants, which is not the case in the argument.

The correct answer is $T(n) = O(n^2)$. Since $T(n) - T(n-10) = n$, which means every time when $n$ increases by 10, the complexity will increase by $n$. Suppose $n$ is a multiple of 10, i.e. $n = 10M$, where $M$ is a positive integer, then

$$T(20) - T(10) = 20 = 10 \times 2$$

$$T(30) - T(20) = 30 = 10 \times 3$$

$$\cdots$$

$$T(n) - T(n-10) = n = 10 \times \frac{n}{10}$$

Then summing up the above equations, we get $T(n) = T(10) + \sum_{i=2}^{M}(10i) = 1 + 0.5(2 + n/10)(n/10 - 1) = O(n^2)$.

## Problem 3

In a set $T$ of $2n-1$ integers, the *EGZ* subset $S \subset T$ is of size exactly $n$ and that the sum of all its elements is a multiple of $n$, i.e., $n \mid \sum_{x \in S} x$. As the Erdös–Ginzburg–Ziv theorem implies that the *EGZ* subset must exist in any set, the remaining task for you is to determine all of the elements in this subset efficiently.

1. **[15 Points]** Suppose $n = 2^{\mathrm{k}}$ is a power of 2. Please design an algorithm to find one such *EGZ* subset for any set of $2n-1$ integers in $o(n^2)$, and prove its correctness and time complexity. Your algorithm should take a set $T$ as input and output one valid *EGZ* subset $S$.

   **Hint:** *Try to solve a few small cases by hand.*

2. **[15 Points]** Suppose $n$ is an arbitrary composite number. Given an oracle subroutine that finds in $O(p)$ time an *EGZ* subset from a set of $2p-1$ integers for any prime number $p$, can you design a polynomial time (w.r.t. $n$) algorithm for the case of any composite $n$? Prove the correctness and time complexity of your algorithm. You may assume that the factorization of n, i.e., all of its prime factors, are given as a part of the input.

**Solution:**

Let us first describe the two typical divide and conquer approach to problem 3.2:

Suppose $p$ is a prime factor of composite number $n$, and let $q = \frac{n}{p}$, which is not necessary a prime number. We can derive $2n - 1 = (2p - 1) \cdot q + q - 1$. This implies that from the original set $T$, one can repeatedly compose a set of size $2q - 1$ for $2p - 1$ times: combine each $q$ with the remaining $q - 1$ numbers. For each of the set of size $2q - 1$, we can extract an EGZ set of size $q$ and the remaining $q - 1$ numbers for recycle —- we reduced the original problem into $2p-1$ subproblems. As these subproblems are solved by recursion, we then have $2p - 1$ number of $q$ elements whose sum is a multiple of $q$. It only remains to find $p$ among $2p - 1$ of these groups that the total sum is a multiple of $n$, or equivalently a multiple of $p$ if only we divide the sum of each group by $q$ – which can be solved by the oracle.

Meanwhile, we can also write $2n - 1 = (2q - 1) \cdot p + p - 1$. So similarly one can repeatedly compose a set of size $2p - 1$ for $2q - 1$ times. For each of the set of size $2p - 1$, we can extract an EGZ set of size $p$ by oracle and the remaining $p - 1$ numbers for recycle. So the remaining problem is to find $q$ among $2q - 1$ of these groups such that their total sum divided by $p$ is a multiple of $q$ – a subproblem to be solved by recursion.

Therefore, in a nutshell, the first approach is to reduce the problem into finding $2p - 1$ number of EGZ set size of $q$, and finally a problem to find EGZ set of size $p$; the second approach is a reduction to find $2p - 1$ number of EGZ set size of $q$, and finally a problem to find EGZ set of size $q$. You may refer to algorithm 1, 2 below for the formal description.

**Algorithm 1** Find-EGZ // approach 1

**Input:** set $\mathcal{T}$, EGZ size $n$, $n = p \cdot q$, where $p$ is a prime number.
**if** $q == 1$ **then**
  **return** Oracle(T)
**end if**
Initialize set $\mathcal{S}, \mathcal{T}'$, map $\mathcal{M}$
**while** $|\mathcal{T}'| \geq 2p - 1$ **do**
  $\mathcal{S}_0 \leftarrow$ Find-EGZ($T[0 : 2q - 1], q$)  // takes $T_1(q)$
  $a \leftarrow \frac{1}{q}$SUM($\mathcal{S}_0$)
  $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{a\}$
  $\mathcal{T} \leftarrow \mathcal{T} \backslash \mathcal{S}_0$
  $\mathcal{M}[a] \leftarrow \mathcal{S}_0$
**end while**
$\mathcal{S}' \leftarrow$ Oracle($\mathcal{T}'$)  // takes $O(p)$
**for** each $e \in \mathcal{S}'$ **do**
  $S \leftarrow \mathcal{S} \cup \mathcal{M}[e]$  // takes $O(q)$
**end for**
**return** $\mathcal{S}$

**Algorithm 2** Find-EGZ // approach 2

**Input:** set $\mathcal{T}$, EGZ size $n$, $n = p \cdot q$, where $p$ is a prime number.
**if** $q == 1$ **then**
  **return** Oracle(T)
**end if**
Initialize set $\mathcal{S}, \mathcal{T}'$, map $\mathcal{M}$
**while** $|\mathcal{T}'| \geq 2q - 1$ **do**
  $\mathcal{S}_0 \leftarrow$ Oracle($T[0 : 2p - 1]$) // takes $O(p)$
  $a \leftarrow \frac{1}{p}$SUM($\mathcal{S}_0$)
  $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{a\}$
  $\mathcal{T} \leftarrow \mathcal{T} \backslash \mathcal{S}_0$
  $\mathcal{M}[a] \leftarrow \mathcal{S}_0$
**end while**
$\mathcal{S}' \leftarrow$ Find-EGZ($\mathcal{T}'$, $q$ ) // takes $T_2(q)$
**for** each $e \in \mathcal{S}'$ **do**
  $S \leftarrow \mathcal{S} \cup \mathcal{M}[e]$ // takes $O(p)$
**end for**
**return** $\mathcal{S}$

The correctness of the two algorithms follows directly from induction. One caveat here is that the division on SUM($\mathcal{S}_0$) is necessary, because there is no guarantee that $p$ is coprime with $q$, i.e., $p$ may not be the unique prime factor.

Time complexity: Denote $T_1(n), T_2(n)$ as time complexity function for approach 1 and 2 respectively to find an EGZ set of size $n$ among $2n - 1$ numbers. On one hand, the algorithm needs to gather at least $n$ numbers for the output set, so we know the lower bound is at least $T_1(n) = T_2(n) \in \Omega(n)$. On the other hand, we know any prime factor $p \geq 2$, so $q \leq \frac{n}{2}$ and we can derive the upper bound for approach 2,

$$T_2(n) = T_2(q) + (2q - 1) \cdot O(p) + p \cdot O(q) = T_2(q) + O(n) \leq T_2(\frac{n}{2}) + \Theta(n) \implies T_2(n) = \Theta(n)$$

More intricate is the upper bound for approach 1, as we cannot directly apply the master's theorem, for $p, q$ are not constant in the recursion. Nevertheless, since $q = n/p$ and $p \geq 2$, we can derive the lower bound of critical exponent, $\log_p(2p-1) > log_p(1)+1 > 1$. This implies that for any fixed prime factor, the recurrence reduction follows the case 1 of master theorem. Meanwhile, since $p^2 > 2p - 1$ when $p \geq 2$, we can derive the upper bound of critical exponent $\log_p(2p - 1) < 2$. So the worst case is when $n = p^k$, and $p$ maximizes $\log_p(2p - 1)$, its time complexity is strictly better than $\Theta(n^2)$.

$$T_1(n) = (2p - 1)T_1(q) + O(q) + q \cdot O(p) = (2p - 1)T_1(q) + O(n) \implies T_1(n) = o(n^2)$$

Essentially, approach 2 achieves the information-theoretical lower bound to solve this EGZ problem. Approach 1 is relatively slower, as it solves too many useless but costly subproblems – finding $2p - 1$ EGZ sets of size $q$, but only about half of them are useful for the final solution.

The problem 3.1 is a special case of problem 3.2, as 3.1 is to have all of its prime factor as 2. Although in 3.1 we do not have oracle for the case when $n = 2$, the task is simply to find two numbers among three that have even sum by brutal force trials in O(1). The time complexity can also be derived through the general formula above, such that $T_1(n) = \Theta(n^{\log_2 3}), T_2(n) = \Theta(n)$

## Problem 4

A polynomial of degree-$n$, $A(x) = \sum_{i=0}^{n-1} a_i x^i$, can be represented as a vector of its coefficients $\boldsymbol{a} = (a_0, a_1, \cdots, a_{n-1}) \in \mathbb{R}^n$.

1. **[10 Points]** Given a polynomial $A(x)$ of degree-$n$ with its representation vector $\boldsymbol{a}$, you want to compute $A(x_0)$ for any given real number $x_0 \in \mathbb{R}$. Suppose that you are only allowed to use the four "basic operations": $+, -, \times, /$, each taking a unit of time. However, you cannot compute $x^n$ in constant time as it requires $n$ multiplications. Design an algorithm to compute $A(x_0)$ in $O(n)$ time. Your algorithm should take $\boldsymbol{a}$ as input and correctly outputs the value $A(x_0)$.

   Moreover, prove that $\Omega(n)$ is the optimal time complexity lower bound — i.e., there is no algorithm that can correctly compute $A(x_0)$ in $o(n)$ time.

   **Solution:** Decompose the polynomial $A(x) = \sum_{i=0}^{n-1} a_i x^i$ into $A(x) = x(\ldots(x(a_{n-1}x + a_{n-2}) + a_{n-3})\ldots a_1) + a_0$, then the computing $A(x)$ can be completed in $O(n)$ time. The algorithm is shown in Algorithm 3.

   To prove the lower bound, assume there is an algorithm that can compute $A(x_0)$ for any $x_0$ in $o(n)$ time. Therefore, this algorithm must be able to compute $A(1) = \sum_{i=0}^{n-1} a_i$ in $o(n)$ time, which is impossible since it has to take $n-1$ addition operations.

**Algorithm 3** Computing $A(x_0)$

**Input:** coefficients $\mathbf{a}$, $x_0$
Initialize $s = 0$, $n = \text{length}(\mathbf{a})$
**for** $i = n-1$ **to** $1$ **do**
  $s = (\mathbf{a}[i] + s)x$
**end for**
$s = s + \mathbf{a}[0]$
**Output:** $s$

2. **[8 Points]** During Lecture 3, we described the Fast Fourier Transform (FFT) algorithm which can be viewed as computing the values of a degree-$(n-1)$ polynomial at $\omega_n^j$ for $j = 0, 1, \cdots, n-1$ (recall $\omega_n = e^{-2\pi i/n}$). The algorithm we described requires $n$ to be a power of 2. In this question, you are asked to show that the algorithm can be adapted to work for any $n$ and any $\omega_m$ where $m = 2^k$ is a power of 2.

   In particular, show how to adapt the FFT algorithm to compute the values of a degree-$(n-1)$ polynomial at $\omega_m^j$ for $j = 0, 1, \cdots, m-1$ for any $n$ and any $m = 2^k$ is a power of 2 ($n$ is not necessarily a power of 2 any more).

   **[Bonus 5 Points]** It turns out that the aforementioned restriction of $m$ to be a power of 2 is *not* needed neither. It requires a much more sophisticated use of the similar Divide-and-Conquer idea we discussed in class. This is not required question for your homework, however you will receive 5 bonus points for solving this problem. [Warning: solving this problem for the sake of points is likely to not worth your time, and is *not recommended*. So do this problem only if you are really interested.]

**Solution:** To use FFT when $n$ is not a power of 2, just add zero coefficients to high-order terms to make the degree of $f(x)$ a power of 2.

For the solution in general case, please check out Cooley–Tukey FFT algorithm online.

3. **[12 Points]** The key idea underlying FFT is to divide a polynomial into odd and even terms. However, this is not the only way to apply the idea of divide-and-conquer. In fact, cutting a polynomial right in the middle appears a more straightforward way. Please design a divide and conquer algorithm for polynomial multiplication using this idea and show that it can still improve upon the time efficiency $O(n^2)$. For simplicity purposes, you may assume $n$ to be a power of 2.

**Hint:** *Try to write* $A(x) = \sum_{i=0}^{n-1} a_i x^i$ *as* $\sum_{i=0}^{(n-2)/2} a_i x^i + \sum_{i=n/2}^{n-1} a_i x^i$ *and see what happens.*

**Solution:**

$A(x) = \sum_{k=0}^{\frac{n-2}{2}} a_k x^k + \sum_{k=n/2}^{n-1} a_k x^k$, $B(x) = \sum_{k=0}^{\frac{n-2}{2}} b_k x^k + \sum_{k=n/2}^{n-1} b_k x^k$. The multiplication of $A(x)$ and $B(x)$ is

$$
\begin{aligned}
C(x) &= A(x)B(x) \\
&= \sum_{k=0}^{(n-2)/2} \sum_{j=0}^{(n-2)/2} a_k b_j x^{k+j} + \sum_{k=0}^{(n-2)/2} \sum_{j=n/2}^{n-1} a_k b_j x^{k+j} \\
&+ \sum_{k=n/2}^{n-1} \sum_{j=0}^{(n-2)/2} a_k b_j x^{k+j} + \sum_{k=n/2}^{n-1} \sum_{j=n/2}^{n-1} a_k b_j x^{k+j} \\
&= \sum_{k=0}^{(n-2)/2} \sum_{j=0}^{(n-2)/2} (a_k b_j + x^{\frac{n}{2}} a_k b_{j+\frac{n}{2}} + x^{\frac{n}{2}} a_{k+\frac{n}{2}} b_j + x^n a_{k+\frac{n}{2}} b_{j+\frac{n}{2}}) x^{k+j} \\
&= \sum_{k=0}^{(n-2)/2} \sum_{j=0}^{(n-2)/2} (a_k b_j + x^{\frac{n}{2}} (a_k b_{j+\frac{n}{2}} + a_{k+\frac{n}{2}} b_j) + x^n a_{k+\frac{n}{2}} b_{j+\frac{n}{2}}) x^{k+j} \\
&= g_0(x) + x^{\frac{n}{2}} (g_1(x) - g_0(x) - g_2(x)) + x^n g_2(x),
\end{aligned}
$$

where we define three functions

$$
\begin{aligned}
g_0(x) &= \sum_{k=0}^{(n-2)/2} \sum_{j=0}^{(n-2)/2} a_k b_j x^{k+j} = (\sum_{k=0}^{(n-2)/2} a_k x^k)(\sum_{j=0}^{(n-2)/2} b_j x^j) \\
&= A_0(x) B_0(x),
\end{aligned}
$$

$$
\begin{aligned}
g_1(x) &= \sum_{k=0}^{(n-2)/2} \sum_{j=0}^{(n-2)/2} (a_k + a_{k+\frac{n}{2}})(b_j + b_{j+\frac{n}{2}}) x^{k+j} \\
&= (\sum_{k=0}^{(n-2)/2} (a_k + a_{k+\frac{n}{2}}) x^k)(\sum_{j=0}^{(n-2)/2} (b_j + b_{j+\frac{n}{2}}) x^j) \\
&= A_1(x) B_1(x),
\end{aligned}
$$

$$g_2(x) = \sum_{k=0}^{(n-2)/2} \sum_{j=0}^{(n-2)/2} a_{k+\frac{n}{2}} b_{j+\frac{n}{2}} x^{k+j} = (\sum_{k=0}^{(n-2)/2} a_{k+\frac{n}{2}} x^k)(\sum_{j=0}^{(n-2)/2} b_{j+\frac{n}{2}} x^j)$$
$$= A_2(x)B_2(x).$$

In effect, we divide the original problem into three sub problems $g_0(x)$, $g_1(x)$, and $g_2(x)$, each solving the multiplication of two polynomials with degree $n/2$. After solving $g_0(x)$, $g_1(x)$, and $g_2(x)$, computing $C(x)$ takes $O(n)$ time. The recursive formula for time complexity is $T(n) = 3T(n/2)+O(n)$. According to the Master Theorem, the time complexity is $T(n) = \Theta(n^{\log_2 3}) = o(n^2)$. The algorithm is shown in Algorithm 4.

**Algorithm 4** PolyMul($A(x)$,$B(x)$,$n$)

**if** n==1 **then**
  Output: $A(x)B(x)$
**end if**
Divide $A(x)B(x)$ into $g_0(x)$, $g_1(x)$, $g_2(x)$
Construct two polynomials $A_0(x)$, $B_0(x)$ from $g_0(x)$
Construct two polynomials $A_1(x)$, $B_1(x)$ from $g_1(x)$
Construct two polynomials $A_2(x)$, $B_2(x)$ from $g_2(x)$
$y_0$=PolyMul($A_0(x)$,$B_0(x)$,$n/2$)
$y_1$=PolyMul($A_1(x)$,$B_1(x)$,$n/2$)
$y_2$=PolyMul($A_2(x)$,$B_2(x)$,$n/2$)
$y = y_0 + x^{n/2}(y_1 - y_0 - y_2) + x^n y_2$
**Output:** $y$