CSOR W4231: Homework 1

Problem 1 (15 points)

(a) (7 points) Graded by Shenzi.

$T(n) = \Theta(n^2)$.
Justification: There are two nested loops here. Since at each iteration of the innermost loop, we do $\Theta(1)$ operations, we have the total number of operations done is Θ of:
$\sum_{i=1}^{n} \sum_{j=i}^{n} 1 = \sum_{i=1}^{n} (n - (i-1)) = \sum_{k=1}^{n} (k) = n(n+1)/2 = n^2/2 + n/2 = \Theta(n^2)$

(b) (8 points) Graded by Ziqing.

$T(n) = \Theta(n^3)$.
Justification: There are three nested loops here. Since at each iteration of the innermost loop, we do $\Theta(1)$ operations, we have the total number of operations done is Θ of:
$\sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} 1 = \sum_{i=1}^{n} \sum_{j=i}^{n} (j-i+1) = \sum_{i=1}^{n} \sum_{m=1}^{n-i+1} m = \sum_{i=1}^{n} (n-i+1)(n-i+2)/2 = \sum_{t=1}^{n} t(t+1)/2 = n(n+1)(n+2)/6 = \Theta(n^3)$

[Note:] A method that may prove easier for more complex sums could be to bound the sum from above and below. If these two bounds both sum to the same Θ time, then you know the sum in question does as well. For example, for part (b), the triple sum is bounded from below by $\sum_{i=1}^{n/3} \sum_{j=2n/3}^{n} (n/3) = (n/3)^3 = n^3/27$, and it is bounded from above by $\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} 1 = n^3$. Therefore it is $\Theta(n^3)$.

Problem 2 (23 points) Graded by Rabia and Niloofar.

The order of the growth of the functions is as follows:

$10 < \log\log n < \log_{10} n = \Theta(\log(n^2)) < (\log n)^2 < \sqrt[4]{n} < n(\log n)^2 < (3n-2)^2 = \Theta(2^{2\log n}) < n^4 < 2^{(\log n)^2} = n^{\log n} < 4^{\sqrt{n}} < \sqrt{2^n} < 2^{3n} < 3^{2n} < n! < n^n$

Justification:

- $10 = o(\log\log n)$ as $\lim_{n\to\infty} \frac{10}{\log\log n} = 0$
- $\log\log n = o(\log_{10} n)$ as $\log_{10} n = \Theta(\log n)$ and $\log\log n = o(\log n)$. This can be proved as $\lim_{n\to\infty} \frac{\log\log n}{\log n} = \lim_{n\to\infty} \frac{1}{\log n}$ [From L'Hospital rule] $= 0$
- $\log_{10} n = \Theta(\log(n^2))$ as $\log(n^2) = 2 * \log n$ and $\log_{10} n = \frac{\log n}{\log 10}$. Hence, $\log(n^2) = 2 * \log 10 * \log_{10} n$
- $\log_{10} n = o((\log n)^2)$ as $\log_{10} n = \Theta(\log n) = o(\log n * \log n)$
- $(\log n)^2 = o(\sqrt[4]{n})$ because $(\log n)^b = o(n^a)$. Proof: Refer class slides
- $\sqrt[4]{n} = o(n(\log n)^2)$ as $\sqrt[4]{n} = o(n) = o(n(\log n)^2)$

- $n(\log n)^2 = o((3n-2)^2)$ as $(3n-2)^2 = 9n^2 - 12n + 4 = \Theta(n^2)$ and $(\log n)^b = o(n^a)$. Thus, $(\log n)^2 = o(n)$ hence $n(\log n)^2 = o((n^2))$.
- $(3n-2)^2 = \Theta(2^{2\log n})$ as $(3n-2)^2 = 9n^2 - 12n + 4 = \Theta(n^2)$ and $2^{2\log n} = 2^{\log n^2} = n^2$.
- $2^{2\log n} = o(n^4)$ as $2^{2\log n} = 2^{\log n^2} = n^2$ and $\lim_{n\to\infty} \frac{n^2}{n^4} = \lim_{n\to\infty} \frac{1}{n^2} = 0$
- $n^4 = o(n^{\log n})$ as $\lim_{n\to\infty} \frac{n^4}{n^{\log n}} = \lim_{n\to\infty} \frac{1}{n^{\log n - 4}} = 0$
- $2^{(\log n)^2} = \Theta(n^{\log n})$ because $2^{(\log n)^2} = 2^{\log(n^{\log n})} = n^{\log n}$
- $2^{(\log n)^2} = o(4^{\sqrt{n}})$ as $\lim_{n\to\infty} \frac{2^{(\log n)^2}}{4^{\sqrt{n}}} = \lim_{n\to\infty} 2^{(\log n)^2 - 2\sqrt{n}} = 0$
- $4^{\sqrt{n}} = o(\sqrt{2^n})$ as $\lim_{n\to\infty} \frac{4^{\sqrt{n}}}{\sqrt{2^n}} = \lim_{n\to\infty} 2^{2\sqrt{n} - \frac{n}{2}} = 0$
- $(\sqrt{2^n}) = o(2^{3n})$ as $\lim_{n\to\infty} \frac{\sqrt{2^n}}{2^{3n}} = \frac{1}{2^{\frac{5n}{2}}} = 0$
- $2^{3n} = o(3^{2n})$ as $\lim_{n\to\infty} \frac{2^{3n}}{3^{2n}} = \lim_{n\to\infty} \frac{8^n}{9^n} = \lim_{n\to\infty} (\frac{8}{9})^n = 0$
- $3^{2n} = o(n!)$ as $\lim_{n\to\infty} \frac{3^{2n}}{n!} = \lim_{n\to\infty} \frac{9^n}{\sqrt{2\pi n}(n/e)^n(1+\Theta(1/n))}$ [By Stirlings Formula] $= \lim_{n\to\infty} \frac{1}{\sqrt{2\pi n}(n/9e)^n(1+\Theta(1/n))}$ $=0$
- $n! = o(n^n)$ as $\lim_{n\to\infty} \frac{n!}{n^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}(n/e)^n(1+\Theta(1/n))}{n^n} = 0$

Problem 3 (27 points) Graded by Ananth and Zhenchen

.

a. (17 points)

In the following algorithm, we assume that the arrays A, B are 1-indexed - that is, they start from an index of 1 and end at n.

---

function `find-100-sum`(A, B)**:**
  $i = 1, j = n$
  while $i \leq n$ *and* $j > 0$ **do**
    if $A[i] + B[j] == 100$ **then**
      return true
    else if $A[i] + B[j] > 100$ **then**
      $j = j - 1$
    else
      $i = i + 1$
    end
  end
  return false

Proof of Correctness:

Loop Invariant (Induction Hypothesis):

Let i, j be the iterators for arrays A, B respectively. The loop invariant is the fact that there does not exist any element of A before $A[i]$ that adds up to 100 with any element of B, and there does not exist any element

of B after $B[j]$ that adds up to 100 with any element of A.
Initialization (Base Case):
$i = 1$, $j = n$. There are no elements of A before $A[1]$ and no elements of B after $B[n]$, so the invariant holds trivially.
Maintenance (Induction Step):
Case 1: $A[i] + B[j] > 100$. Since we decrement j by 1 for the next iteration, we need to check that $B[j]$ does not add up to 100 with any element $A[k]$ of A. The case of $k < i$ is taken care of by the induction hypothesis. If $k \geq i$, $A[k] \geq A[i]$ since A is sorted, so $A[k] + B[j] \geq A[i] + B[j] > 100$. So, for every value of k, the loop invariant (induction hypothesis) holds true.
Case 2: $A[i] + B[j] < 100$. Since we increment i by 1 for the next iteration, we need to check that $A[i]$ does not add up to 100 with any element $B[l]$ of B. The case of $l > j$ is taken care of by the induction hypothesis. If $l \leq j$, $B[l] \leq B[j]$ since B is sorted, so $A[i] + B[l] \leq A[i] + B[j] < 100$. So, for every value of l, the loop invariant holds true.
Termination:
If the algorithm returns false, then either i has iterated the full length of A, i.e. $i > n$, or j has iterated the full length of B, i.e. $j = 0$. The loop invariant implies that there does not exist any pairing of numbers that add up to 100, so the algorithm has returned the correct value.
If the algorithm returned true, then i and j must be at indices where $A[i] + B[j] = 100$, so the algorithm has returned the correct value. Thus, the algorithm is correct in all cases.
Running Time:
At most, the algorithm will iterate through both arrays once, covering $2n$ elements. Every iteration takes constant time. Therefore, the (worst-case) running time of this algorithm is $\Theta(n)$.

b.(10 points)

---

function `find-100-sum-2(`C`)`**:**
  MERGESORT(C)
  $i = 1, j = n$
  while $i < j$ **do**
    if $C[i] + C[j] == 100$ **then**
      return true
    else if $C[i] + C[j] > 100$ **then**
      $j = j - 1$
    else
      $i = i + 1$
    end
  end
  return false

Proof of Correctness:
We know that mergesort is a correct algorithm that sorts an array in $\Theta(nlogn)$ time. Once C is sorted, the algorithm for this problem is very similar to that of part (a) - except for the fact that C replaces both A and B, and that we stop the loop if the iterators i, j satisfy $i \geq j$. To see that it is safe to stop the loop at that point and return false, note that the loop invariant of part (a) implies that no element $C[k]$ with $k < i$ or $k > j$ adds up to 100 with any other element of C. If $i \geq j$ this implies that C does contain two elements that sum to 100.
Running Time:
Θ(nlogn), since the slowest part of the algorithm is mergesort. The rest of the algorithm runs in linear time ($O(n)$).

Problem 4 (35 points) Graded by Aditya and Malhar.

1. (10 points)

Suppose $a = k+1$ and $b = k$ for some integer $k > 0$. Then, $n = (\lfloor \log a \rfloor + 1) + (\lfloor \log b \rfloor + 1) = (\lfloor \log (k+1) \rfloor + 1) + (\lfloor \log k \rfloor + 1) \leq 2 \log k + 3$. Thus, $\log k \geq (n/2) - 1.5$.
However, since $k+1$ and k are coprime (i.e. their only common divisor is 1), the while instruction will be executed k times. Since $k = 2^{\log k}$, we have that $k = \Omega(2^{n/2})$ and the algorithm runs in time exponential in n, not polynomial.

2. (25 points)

a. (10 points)

Initialization: Clearly, since x is defined as $\max(a, b)$ and y as $\min(a, b)$, (i) is true before the first iteration. (ii) is also trivially true since either $x = a$ and $y = b$ or $x = b$ and $y = a$.
Maintenance: We will first show that each iteration maintains (i). Assuming $x \geq y$ at the previous iteration this is fairly obvious. Indeed, since $r = x \mod y < y$, it must be that $x \geq y$ after the next iteration as well since x becomes y and y becomes r.
We now show that each iteration maintains (ii). Assuming $c|a$ (c divides a) and $c|b$ iff $c|x$ and $c|y$, we just have to show that $c|x$ and $c|y$ iff $c|(x \mod y)$ and $c|y$ at every iteration. Recall that if $c|x$ and $c|y$, then c divides any integer combination $kx+ly$ of x and y, where k, l are integer: indeed, $c|x$ and $c|y$ mean that $x = ic$ and $y = jc$ for some integer i, j, hence $kx+ly = kic+ljc = (ki+lj)c$. Since $(x \mod y) = x - \lfloor \frac{x}{y} \rfloor y$, $c|x$ and $c|y$ implies that $c|(x \mod y)$ and $c|y$. Conversely, if $c|(x \mod y)$ and $c|y$, then since $x = (x \mod y) + \lfloor \frac{x}{y} \rfloor y$, we also have that $c|x$ and $c|y$. Therefore, each iteration maintains (ii).
Termination: The loop always terminates because y eventually decreases to 1 if it does not divide x before that and 1 divides any number. At the end of the loop, we have y such that $y|x$, and trivially also $y|y$. So y is a common divisor of both (x, y) and (a, b).

b. (5 points)

From the termination step above, it is easy to see that y is the gcd of a and b. Indeed, since every divisor of both a and b is also a divisor of x and y at termination by (ii), and since y divides both x, y and clearly there is no common divisor of x and y greater than $\min(x, y) = y$, it follows that the greatest common divisor of both (x, y) and (a, b) must be y.

c. (10 points)

We first show that if $x \geq y$, then $(x \mod y) \leq x/2$. Clearly, if $x/2 \leq y < x$, then $x \mod y = x - y \leq x/2$. If $0 < y < x/2$, then $x \mod y < y \leq x/2$.
Therefore, since x and y are switched at each iteration, both x and y must be at least divided by two after every 2 iterations. In other words, since y looses at least 1 bit after every 2 iterations, the algorithm must terminate after at most $2n$ iterations. Thus, Euclid's algorithm performs $O(n)$ operations.