Assignment 4 (SOLUTIONS)

Q1. [10 Points] Multiplying Upper-Triangular Matrices *(maximum 3 pages)*

```
Algorithm 1 multiplyTT($A$ : UT matrix, $B$ : UT matrix) → UT matrix
Require: $A$ and $B$ are two upper-triangular matrices.
Ensure: Outputs $A \cdot B$.
1: if $n = 1$ then return $A \cdot B$
2: end if
3: $M_1 \leftarrow$ multiplyTT($A_{11}, B_{11}$)
4: $M_2 \leftarrow$ multiplyTM($A_{11}, B_{12}$) + multiplyMT($A_{12}, B_{22}$)
5: $M_3 \leftarrow$ multiplyTT($A_{22}, B_{22}$)
6: return $\begin{bmatrix} M_1 & M_2 \\ 0 & M_3 \end{bmatrix}$
```

```
Algorithm 2 multiplyTM($A$ : UT matrix, $B$ : matrix) → matrix
Require: $A$ is an upper-triangular matrix and $B$ is a matrix.
Ensure: Outputs $A \cdot B$.
1: if $n = 1$ then return $A \cdot B$
2: end if
3: $M_1 \leftarrow$ multiplyTM($A_{11}, B_{11}$) + multiply($A_{12}, B_{21}$)
4: $M_2 \leftarrow$ multiplyTM($A_{11}, B_{12}$) + multiply($A_{12}, B_{22}$)
5: $M_3 \leftarrow$ multiplyTM($A_{22}, B_{21}$)
6: $M_4 \leftarrow$ multiplyTM($A_{22}, B_{22}$)
7: return $\begin{bmatrix} M_1 & M_2 \\ M_3 & M_4 \end{bmatrix}$
```

In class, we saw Karatsuba's algorithm which was a way to speed up the computation of $a \cdot b$ for two n-bit integers a and b. In this problem, we analyse an algorithm for multiplying two *upper-triangular*[1] matrices abbreviated *UT matrices*. See Algorithm 1. The standard approach for multiplying two matrices is shown below for two 2×2 matrices.

$$A \cdot B = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \cdot \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} = \begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{bmatrix}$$

For ease of explanation, if the inputs A and B to Algorithm 1, Algorithm 2, and Algorithm 3 have dimension $n \times n$, then let n be a power of two. In practice, we can always pad the matrix with rows and columns of all zeros so that this is the case.

Note that the notation $A[i : j, \ell : k]$ represents the submatrix of A consisting of the rows i to j inclusive and columns ℓ to k inclusive. For A and B, let $A_{11} = A[1 : n/2, 1 : n/2]$, $A_{12} = A[1 :$

[1] A square matrix A is *upper-triangular* if all entries $a_{i,j}$ are equal to zero for $i > j$ (the top left entry of A is $a_{1,1}$ the bottom right entry is $a_{n,n}$). Here are some upper-triangular matrices:

$$A_1 = \begin{bmatrix} 5 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 5 & 1 \\ 0 & 2 \end{bmatrix} \qquad A_3 = \begin{bmatrix} 0 & 3 & 1 \\ 0 & 4 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Algorithm 3 multiplyMT(A : matrix, B : TU matrix) $\to$ matrix

```
Require: $A$ is a matrix and $B$ is an upper-triangular matrix.
Ensure: Outputs $A \cdot B$.
1: if $n = 1$ then return $A \cdot B$
2: end if
3: $M_1 \leftarrow$ multiplyMT($A_{11}, B_{11}$)
4: $M_2 \leftarrow$ multiply($A_{11}, B_{12}$) + multiplyMT($A_{12}, B_{22}$)
5: $M_3 \leftarrow$ multiplyMT($A_{21}, B_{11}$)
6: $M_4 \leftarrow$ multiply($A_{21}, B_{12}$) + multiplyMT($A_{22}, B_{22}$)
7: return $\begin{bmatrix} M_1 & M_2 \\ M_3 & M_4 \end{bmatrix}$
```

$n/2, (n/2+1) : n]$, $A_{21} = A[(n/2+1) : n, 1 : n/2]$, and $A_{22} = A[(n/2+1) : n, (n/2+1) : n]$. B_{ij} are defined similarly for $i, j \in [2]$.

The time it takes to multiple two constant dimension matrices is constant time.

a. Suppose that the time it takes to multiply two numbers or add two numbers is $O(1)$. What is the asymptotic running time of the standard algorithm for multiplying two n by n upper-triangular matrices?

Solution. Note that every off-diagonal entry below the main diagonal (i.e., entries (i, j) where $i > j$) is equal to zero so it suffices to compute every entry on or above the diagonal (i.e., entries (i, j) where $i \leq j$). There are $\Theta(n^2)$ such entries. Further, computing each entry takes $O(n)$ time (take product of n pairs of pairs of numbers then add them) and for $\frac{n^2}{4}$ entries — namely those in the top right corner — at least $n/2$ time. It follows that the total running time is $\Theta(n^3)$.

[1 Mark] Correct asymptotic running time and some reasonable justification.

b. Prove the correctness of Algorithm 1 assuming the correctness of multiply (it takes as input two $n \times n$ matrices A and B and the post-conditions is that it correctly returns $A \cdot B$). You only need to prove *one* of Algorithm 2 and Algorithm 3 correct, but not both.

Proof. We prove the correctness of multiplyMT and then, assuming the correctness of multiplyMT and multiplyTM (the correctness of multiply is given), we prove the correctness of multiplyTT.

Let $P_{MT}(n)$ be the predicate that if A and B are matrices of dimension $n \times n$ and satisfies the preconditions of multiplyMT, then multiplyMT(A, B) will return their product. We show that $P_{MT}(n)$ is true for all $n \in \mathbb{N}$ and $n \geq 1$ (you could also do this by structural induction).

In the base case multiplyMT clearly outputs the correct answer on line 1.

In the inductive step suppose for some fixed $k \geq 2$, $P_{MT}(1) \wedge \cdots \wedge P_{MT}(k-1)$ is true. We want to show that $P_{MT}(k)$ is true. By IH, we have that $M_1 = A_{11} \cdot B_{11}$, $M_3 = A_{21} \cdot B_{11}$, multiplyMT($A_{12}, B_{22}$) returns $A_{12} \cdot B_{22}$ and multiplyMT(A_{22}, B_{22}) returns $A_{22} \cdot B_{22}$. Note that by block multiplying (you could also do this entry-wise), we have that

$$A \cdot B = \begin{bmatrix} A_{11} \cdot B_{11} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{12} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix}$$

as B_{21} is the all zeros matrix. These are exactly the matrices M_1, M_2, M_3, M_4.

Thus, by the principles of mathematical induction $P_{MT}(n)$ is true for all $n \in \mathbb{N}$ with $n \geq 1$.

The proof of the correctness of multiplyTT is very similar. Here we will only touch upon the highlights, *but your solution should be more complete.* In particular, if we perform block multiplication on two triangular matrices, we have that,

$$A \cdot B = \begin{bmatrix} A_{11} \cdot B_{11} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ 0 & A_{22} \cdot B_{22} \end{bmatrix}$$

where $A_{11} \cdot B_{12}$ is an upper triangular matrix multiplying another matrix and $A_{12} \cdot B_{22}$ is a matrix multiplying an upper triangular matrix. We use the inductive hypothesis to guarantee the correctness of multiplyTT(A_{11}, B_{11}) and multiplyTT(A_{22}, B_{22}) and we use the correctness of multiplyTM on the product $A_{11}, \cdot B_{12}$ and multiplyMT on the product $A_{12} \cdot B_{22}$. Together we have that the returned matrix is indeed $A \cdot B$.

[6 Marks] 3 marks for each of two proofs (multiplyMT or multiplyTM and multiplyTT). Among these, 2 marks for the inductive step. □

c. Read about Strassen's algorithm. Compute the asymptotic running time of Algorithm 1 assuming that multiply on lines 3 and 4 of Algorithm 2 and lines 4 and 5 of Algorithm 3 calls Strassen's algorithm.

Solution. First one should note that the time necessary to multiply two $n \times n$ matrices with Strassen's algorithm, denoted $T_S(n)$ is $\Theta\left(n^{\log_2 7}\right)$. Then we need to compute the running time of multiplyMT and multiplyTM. Since they pretty much have the same running times, we will only compute the running time for the former. First we express the running time of multiplyMT on two $n \times n$ matrices with the recurrence relation

$$T_{MT}(n) = 4 \cdot T_{MT}\left(\frac{n}{2}\right) + 2T_S\left(\frac{n}{2}\right) = 4 \cdot T_{MT}\left(\frac{n}{2}\right) + 2\Theta\left(\frac{n}{2}\right)^{\log_2 7}.$$

We will apply the master method to this recurrence relation, with $a = 4$, $b = 2$ and $f(n) = cn^{\log_2 7}$ for some constant c. Note that $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for $\epsilon > 0$ since there exists a $\delta > 0$ such that $\log_2 4 + \delta = \log_2 7$. It follows that $T_{MT}(n) = \Theta(f(n))$.

Finally, we compute the running time of multiplyTT. Note that the recurrence relation for its running time on two input matrices of dimension $n \times n$ is

$$T_{TT}(n) = 2T_{TT}\left(\frac{n}{2}\right) + T_{TM}\left(\frac{n}{2}\right) + T_{MT}\left(\frac{n}{2}\right) = T_{TT}(n) = 2T_{TT}\left(\frac{n}{2}\right) + cn^{\log_2 7}$$

for some constant c (possibly different from before). Again, applying the Master Method, this case is *leaf dominant* so we have $T_{TT}(n) = \Theta\left(n^{\log_2 7}\right)$.

[3 Marks] One mark for stating the running time of Strassen's algorithm. One mark for computing the running time of multiplyMT or multiplyTM. One mark for computing the asymptotic running time of multiplyTT.

Q2. [10 Points] Candy Claw Machine *(maximum 3 pages)*

You are playing a crane game to win as many tasty candies as possible. There are n candies in a row on a conveyor belt and you assigned a value $v_i \in \mathbb{R}$ to the candy at position i (note the value can be negative). At each time unit the conveyor belt will move to the left by one candy. You can lower the claw at time t and retract the claw at time s where $0 \le t \le s \le n$ (when $t = s = n$, you will not get any candies). *You can only lower and raise the claw once.*

You *collect* all candies at positions t through s inclusive (e.g. if $t = 0$ and $s = n - 1$ then you collect all the candies). The *value* of the candies you collect is $\sum_{i=t}^{s} v_i$. Come up with a *recursive* algorithm to determine when you should lower and raise the claw to collect the most valuable candies. If there are multiple time intervals obtaining the maximum value, you can output any.

Your algorithm should run in $O(n)$ time. If it runs in $O(n \log n)$ and you can prove this, you will get part marks.

As an example, suppose that you have the candies on the conveyor belt have the following values:

$$[1, 2, -9, 3, 3, 1, -2, 1]$$

Then the most valuable candies can be obtained by lowering the claw at time 3 (picking up the first three value candy) and raising it at time 5 (picking up the one value candy). The total value of this collection is 7.

If all the candies have negative value, as is the case for this conveyor belt:

$$[-1, -4, -2, -2, -3],$$

you can lower and raise the claw at time 5 to collect no candies.

a. Write the pseudo-code in Algorithm 4.

As many of you have noticed, this is precisely the *maximum sum sub-array* problem. Algorithm 4 take as input the original list and the interval $[i, j]$ that we are considering. Instead of returning a list, we will return a structure with several parameters called a *decorated list*. Let a *decorated list* be a structure with the following parameters:

(a) L: the associated list
(b) t_ℓ and t_r: the left and right indices of the maximum sum sub-array of L
(c) p: index of the end-point of the maximum *prefix*[2] of L
(d) s: index of the end-point of the maximum *suffix*[3] of L
(e) v_t, v_p, v_s, v_m: the value of the sum of the entries in the maximum sub-array, prefix, suffix, and the sum of all the entries in L respectively.

We access the parameters with dot-notation, e.g., on decorated list $\mathcal{A}$, to access the associated list we write $\mathcal{A}.L$. The pseudo-code is shown in Algorithm 4. To find the answer to the entire array A of length n, we call $\mathsf{Claw}(A, 0, n-1)$ to obtain $\mathcal{A}$. If $\mathcal{A}.v_t < 0$, then we return the indices (n, n) (i.e., pick up no candies), otherwise we return $(\mathcal{A}.t_\ell, \mathcal{A}.t_r)$.

[4 Marks] The algorithm should be recursive (lose two marks if it is not) and run in time $O(n)$ (lose two marks if is runs in $O(n \log n)$). Give no marks if it runs in time $\omega(n \log n)$.

[2] A prefix of a list $A = [a_0, ..., a_{n-1}]$ is any contiguous sub-array starting from a_0, e.g., $[a_0, ..., a_4]$ is a prefix of A which ends at index 4.

[3] A suffix of a list $A = [a_0, ..., a_{n-1}]$ is any contiguous sub-array ending at a_{n-1}, e.g., $[a_4, ..., a_{n-1}]$ is a suffix of A which ends at index 4.

Algorithm 4 Claw(A, i, j)

```
Require: List $A$ and indices $i \leq j$.
Ensure: Decorated list $\mathcal{A}$ with associated list $A[i,j]$ and all correct values for $A[i,j]$ filled in.
 1: $\mathcal{A}.L \leftarrow A[i,j]$    ▷ only pointers to end-points of the interval need to be preserved
 2: if $j = i$ then    ▷ base case when $A[i,j]$ is a single entry
 3:     $\mathcal{A}.v_m \leftarrow A[i]$
 4:     if $A[i] < 0$ then
 5:         $\mathcal{A}.t_\ell, \mathcal{A}.t_r, \mathcal{A}.p, \mathcal{A}.s \leftarrow (n, n, i+1, i-1)$    ▷ indices of the empty set w.r.t. $A[i,i]$
 6:         $\mathcal{A}.v_t, \mathcal{A}.v_s, \mathcal{A}.v_p \leftarrow (0, 0, 0)$    ▷ value of the empty set
 7:     else
 8:         $\mathcal{A}.t_\ell, \mathcal{A}.t_r, \mathcal{A}.p, \mathcal{A}.s \leftarrow (i, i, i, i)$
 9:         $\mathcal{A}.v_t, \mathcal{A}.v_s, \mathcal{A}.v_p \leftarrow (A[i], A[i], A[i])$
10:     end if
11:     return $\mathcal{A}$
12: end if
13: $m \leftarrow (j+i)/2$
14: $\mathcal{B} \leftarrow$ Claw$(A, i, m)$    ▷ recurse on the left half of $A[i,j]$
15: $\mathcal{C} \leftarrow$ Claw$(A, m+1, j)$    ▷ recurse on the right half of $A[i,j]$
16: $\mathcal{A}.v_m \leftarrow \mathcal{B}.v_m + \mathcal{C}.v_m$
17: $\mathcal{A}.v_t \leftarrow \max(\mathcal{B}.v_t, \mathcal{C}.v_t, \mathcal{B}.v_s + \mathcal{C}.v_p)$
18: if $\mathcal{A}.v_t = \mathcal{B}.v_t$ then    ▷ fix the index of the sub-array in $A[i,j]$ with maximum sum
19:     $\mathcal{A}.t_\ell, \mathcal{A}.t_r \leftarrow (\mathcal{B}.t_\ell, \mathcal{B}.t_r)$
20: else if $\mathcal{A}.v_t = \mathcal{C}.v_t$ then
21:     $\mathcal{A}.t_\ell, \mathcal{A}.t_r \leftarrow (\mathcal{C}.t_\ell, \mathcal{C}.t_r)$
22: else
23:     $\mathcal{A}.t_\ell, \mathcal{A}.t_r \leftarrow (\mathcal{B}.s, \mathcal{C}.p)$
24: end if
25: $\mathcal{A}.v_p \leftarrow \max(\mathcal{B}.v_p, \mathcal{B}.v_m + \mathcal{C}.v_p)$
26: if $\mathcal{A}.v_p = \mathcal{B}.v_p$ then    ▷ fix the index for the prefix of $A[i,j]$
27:     $\mathcal{A}.p \leftarrow \mathcal{B}.p$
28: else
29:     $\mathcal{A}.p \leftarrow \mathcal{C}.p$
30: end if
31: $\mathcal{A}.v_s \leftarrow \max(\mathcal{C}.v_s, \mathcal{B}.v_s + \mathcal{C}.v_m)$
32: if $\mathcal{A}.v_s = \mathcal{C}.v_s$ then    ▷ fix the index of the suffix of $A[i,j]$
33:     $\mathcal{A}.s \leftarrow \mathcal{C}.s$
34: else
35:     $\mathcal{A}.s \leftarrow \mathcal{B}.s$
36: end if
37: return $\mathcal{A}$
```

b. Prove the correctness of your algorithm. Remember to *state the post-condition.*

Proof. The post-condition is stated in the pseudo-code, so we will proceed to the proof.

Our recursively defined set will be all singleton elements $A[i]$ for $i = 0, ..., n-1$ and, for two sub-arrays sub-arrays $A[i,j], A[j,k] \subseteq A$, $A[i,k]$ is also in our set.

Our predicate will be $P(A[i,j])$: the decorated list $\mathcal{A} \to \mathsf{Claw}(A,i,j)$ have filled parameters which satisfy their definition for $A[i,j]$, i.e., $\mathcal{A}.v_t$ stores the maximum value of any sub-array in $A[i,j]$, $(\mathcal{A}.t_\ell, \mathcal{A}.t_r)$ is the left and right indices of a sub-array which achieves $\mathcal{A}.v_t$, and so on for all the other parameters.

In the base case where $i = j$, we have that the outputs on lines 2 to 12 are correct by inspection.

In the inductive step, suppose we have that we have $P(A[i,m]) \wedge P(A[m+1,j])$ are true, we want to show that $P(A[i,j])$ is true. By IH, we have that $\mathcal{B} \to \mathsf{Claw}(A,i,m)$ and $\mathcal{C} \to \mathsf{Claw}(A,m+1,j)$ are decorated lists which satisfy their post-condition. Let us consider each parameter of $\mathcal{A}$ in-turn.

- L By definition $\mathcal{A}$ is associated with sub-array $A[i,j]$.
- v_m Since $\mathcal{B}.v_m$ and $\mathcal{C}.v_m$ are the sums of the left and right halfs of $A[i,j]$ respectively, $\mathcal{A}.v_m = \mathcal{B}.v_m + \mathcal{C}.v_m$.
- v_p, p The value of the prefix of $A[i,j]$ with the largest sum is *either* a prefix of $A[i,m]$ *or* a prefix containing all of $A[i,m]$ and the maximum prefix of $A[m+1,j]$. The index of this interval can be assigned accordingly.
- v_s, s The value of the suffix of $A[i,j]$ with the largest sum is *either* a suffix of $A[m+1,j]$ *or* a suffix containing all of $A[m+1,j]$ and the maximum suffix of $A[i,m]$. The index of this interval can be assigned accordingly.
- v_t, t_ℓ, t_r Very similar to the above. The sub-array with the maximum sum is either entirely in $A[i,m]$, entirely in $A[m+1,j]$, or is composed of the maximum suffix of $A[i,m]$ the the maximum suffix of $A[m+1,j]$. Indices can be assigned accordingly.

It follows by structural induction that $P(A[i,j])$ is true for every sub-array of A, and in-particular, $A[0,n-1] = A$. □

[4 Marks] Grade the proof-of-correctness independent of the running time of the algorithm. As long as the solution properly analyzes the given pseudo-code it can get full marks.

c. Evaluate the running time of your algorithm and justify your answer.

Solution. The asymptotic running time of our algorithm is $\Theta(n)$. To see this we will justify that the running time of our algorithm is defined by the recurrence relation

$$T(n) = 2T\left(\frac{n}{2}\right) + c$$

for some constant c. Note that the base case and initial setup (lines 2 to 13) take constant time. Then, there are two recursive calls to Claw on inputs half the size which satisfy the pre-conditions (lines 14 and 15). The remainder of the algorithm also runs in constant time as we are simply accessing values that we have previously stored.

By the Master Method, we see that $T(n) = \Theta(n)$.

[2 Marks] One point for the correct asymptotic run time for the pseudo-code. *One point for the justification.*