Quiz 1

- Do not open this quiz booklet until you are directed to do so.
- This quiz ends at 3:55 P.M.
- When the quiz begins, write your name on the top of ***EVERY*** page in this quiz booklet, because the pages will be separated for grading.
- Write your solutions in the space provided. If you need more space, write on the *back* of the sheet containing the problem. Do not put part of the answer to one problem on the back of the sheet for another problem.
- Plan your time wisely. Do not spend too much time on any one problem. Read through all of them first and attack them in the order that allows you to make the most progress.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- When describing an algorithm, describe the main idea in English. Use pseudocode only to the extent that it helps clarify the main ideas.
- Good luck!

Problem	Points	Grade
1	16	
2	30	
3	19	
4	15	
Total	80	

Name: ____________________

Please circle your TA's name and recitation:
Nitin Brian Mihai Yoav

10am 11am 12pm 1pm 2pm

Problem 1. Recurrences [**16 points**]

Solve the following recurrences (provide only the $\Theta()$ bounds). You can assume $T(n) = 1$ for n smaller than some constant in all cases. You *do not* have to provide justifications, just write the solutions.

- $T(n) = 9T(n/3) + n^{1.1}$

  `Answer:` $\Theta(n^2)$
  `Master Method, case 1.`

- $T(n) = T(9n/10) + \log n$

  `Answer:` $\Theta(\log^2 n)$
  `Extended Master Method, case 2. Alternatively, you can expand the recurrence out into a summation` $T(n) = \log n + \log \frac{9}{10}n + \log(\frac{9}{10})^2 n + \ldots + \Theta(1)$ `and conclude that the summation behaves asymptotically as` $\Theta(\log^2 n)$.

- $T(n) = T(\sqrt{n}) + \log n$

  `Answer:` $\Theta(\log n)$
  `This can be solved with a change of variables followed by the Master Method. Alternatively, by expanding the recurrence out:` $T(n) = \log n + \frac{1}{2}\log n + \frac{1}{4}\log n + \ldots + \Theta(1) = \Theta(\log n)$.

- $T(n) = 4T(n/3) + n^2$

  `Answer:` $\Theta(n^2)$
  `Master Method, case 3.`

Problem 2. True or False, and Justify [**30 points**] (6 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false, respectively. If the statement is correct, briefly state why. If the statement is wrong, explain why. Your justification is worth more points than your true-or-false designation.

T F The solution to the recurrence

$$T(n) = 2^n T(n-1)$$

is $T(n) = \Theta((\sqrt{2})^{n^2+n})$ (assume $T(n) = 1$ for n smaller than some constant c).

TRUE. `Assume e.g., that` $T(0) = 1$. `By substitution we get`

$$\begin{aligned} T(n) &= 2^n \cdot 2^{n-1} \cdot \ldots 2^0 \\ &= 2^{n+(n-1)+(n-2)+\ldots+1} \\ &= 2^{(n+1)n/2} \\ &= (\sqrt{2})^{n^2+n} \end{aligned}$$

For the curious: assuming that $T(n) = 1$ for $n \leq c$ for some constant c works fine as well. The only difference is that the sum of the decreasing terms in the exponent ends at $c+1$, not 0. This changes the solution by at most a multiplicative factor of $2^{O(c^2)}$, which is constant if c is constant.

T F There exists a comparison-based sorting algorithm that can sort any 4-element array using at most 5 comparisons.

TRUE. `For example, one can use merge sort, which performs:`

- `1 comparison to sort the left subarray of size 2`
- `1 comparison to sort the right subarray of size 2`
- `3 comparisons to merge the two sorted arrays of size 2 each`

`For completeness, here is a popular solution that is NOT correct:`

> `The logarithm of 4! is less than 5, therefore there is a comparison-based algorithm that sorts 4 elements using 5 comparisons.`

`This reasoning is not correct.` $\log(n!)$ `is the *lower bound* on the number of comparisons needed to sort` n `elements. So, this argument only allows us to conclude that *at least 5* comparisons are needed, which is not sufficient to determine if the answer is TRUE or FALSE.`

T F Checking if there is a pair of equal elements in an array $A[1\ldots n]$ requires $\Omega(n^2)$ time in the comparison-based model, because we need to test equality for every pair of elements.

FALSE. One can solve the problem in $O(n \log n)$ time, by sorting the array A and then scanning it for a pair of adjacent equal elements.

T F Consider an implementation of Paranoid Quicksort which accepts a partition of an array $A[1\ldots n]$ as balanced only if the lengths of the two subarrays resulting from the partition differ by at most ± 3. The expected running time of this Quicksort implementation is $O(n \log n)$.

FALSE. Consider the case where all elements are distinct. The probability that a random partition is ± 3 balanced is at most $7/n$. Thus, the expected number of partitions generated until the first balanced one is created is at least $n/7$. Since each partition takes $\Omega(n)$ time, the total running time is $\Omega(n^2)$.

T F An array $A[1 \dots n]$ is called *bitonic*, if there exists t such that $A[1 \dots t]$ is sorted in the increasing order, and $A[t \dots n]$ is sorted in the decreasing order.
There is a linear time comparison-based algorithm for sorting bitonic arrays.

TRUE. It suffices to merge arrays $A[1 \dots t]$ and $(A[t+1 \dots n])^R$, where B^R denotes an array B in the reverse order. Merging two sequences of length n can be done in $O(n)$ time.

T F An array $A[1 \dots n]$ is called *pytonic*, if every element stored at even position is smaller than its two neighboring elements (formally, for every integer $k > 0$, $A[2k-1] > A[2k]$ and $A[2k] < A[2k+1]$). E.g., the array $< 5, 1, 6, 2, 7, 3 >$ is pytonic.

There is a linear time comparison-based algorithm for sorting pytonic arrays.

FALSE. The simplest proof is by contradiction. Assume there is an $O(n)$ algorithm Alg for sorting pytonic arrays. Then, we could design an algorithm that sorts any sequence $a_1 \dots a_n$ in $O(n)$ time as well. The algorithm would perform the following steps:

- Create a pytonic array with elements $a_1, -\infty, a_2, -\infty, \dots, a_n, -\infty$, where the symbol $-\infty$ denotes an element that is smaller than any other element.
- Sort the latter sequence using Alg, in $O(n)$ time.
- Report the last n elements of the sorted sequence.

Since there is an $\Omega(n \log n)$ lower bound for comparison-based sorting of arbitrary sequences, the contradiction follows.

Alternatively, one could reproduce the counting argument used to show the $\Omega(n \log n)$ lower bound for sorting of general inputs, but restricted to sequences of the form $a_1, -\infty, a_2, -\infty, \dots, a_n, -\infty$.

Problem 3. Superselection **[19 points]**

Prof. Noitceles Repus proposes the following divide-and-conquer algorithm for selection, that he calls SUPERSELECTION. Suppose we have an array $A[1 \ldots n]$ of distinct elements and we want to select the element with rank i. We first apply SUPERSELECTION recursively on $A[1 \ldots n/2]$ to compute its median M_1. Then we apply SUPERSELECTION on $A[n/2+1 \ldots n]$ to compute its median M_2. Let $m = \min(M_1, M_2)$ and $M = \max(M_1, M_2)$, We use m and M to partition our array A into three parts:

1. Part B: contains all elements $< m$
2. Part C: contains all elements in $[m, M]$
3. Part D: contains all elements $> M$

Finally, we apply SUPERSELECTION recursively on one of the parts B,C or D in an "appropriate" way.

The following (incomplete) pseudocode defines the algorithm more formally:

Superselection$(A, start, end, rank)$
1 **if**$(start = end)$
2   **return** $A[start]$
3 $M_1 \leftarrow$ **Superselection**$(A, start, \lfloor (end + start)/2 \rfloor, \lfloor (end - start)/4 \rfloor)$
4 $M_2 \leftarrow$ **Superselection**$(A, \lfloor (end + start)/2 \rfloor + 1, end, \lfloor (end - start - 1)/4 \rfloor)$
5 $m \leftarrow$ **min**(M_1, M_2)
6 $M \leftarrow$ **max**(M_1, M_2)
7 $(B_{end}, C_{end}) \leftarrow$ **3Partition**(A, m, M)
8 $newstart \leftarrow \ldots$; $newend \leftarrow \ldots$; $newrank \leftarrow \ldots$
9 **return Superselection**$(A, newstart, newend, newrank)$

(a) **[4 points]** Specify what "appropriate" means. I.e., replace line 8 in the above code with new code, that properly specifies the variables *newstart*, *newend* and *newrank*.

Solution: Line 8 should be replaced with the following lines of code:

$rank' \leftarrow rank + start - 1$
if $start \leq rank' \leq B_{end}$ $\triangleright$ Element we seek is in B
    $newstart \leftarrow start$
    $newend \leftarrow B_{end}$
    $newrank \leftarrow rank$
elseif $B_{end} + 1 \leq rank' \leq C_{end}$ $\triangleright$ Element we seek is in C
    $newstart \leftarrow B_{end} + 1$
    $newend \leftarrow C_{end}$

$newrank \leftarrow rank - (B_{end} - start + 1)$

else $\triangleright$ Element we seek is in D

$newstart \leftarrow C_{end} + 1$

$newend \leftarrow end$

$newrank \leftarrow rank - (C_{end} - start + 1)$

A common mistake in the solutions was to fail to realize that *rank* was an offset from *start* and yet modify *rank* in the code.

Another common mistake was abusing m as both an index and a value.

We did not worry about 'off-by-one' issues, since it wasn't necessarily clear if arrays start at 0, or if *rank* of 1 was the first or second element in the sorted list.

(b) [**4 points**] Argue that the partitioning procedure can be implemented in linear time.

Solution: There are several ways to argue this. Generally they comprise of providing a description of how one would do so.

- Run a partition algorithm around m. Since M is bigger than m, it will be in the array to the right of m, along with all elements that are greater than M. Now just run a standard partition AGAIN, but on the subarray composed only of the elements greater than m. This is two invocations of the partition algorithm from class, which runs in $\Theta(n)$.
- Another approach is to actually specify a new partitioning algorithm. Here we need to keep track of three pointers into array A, and deal with the multiple scenarios that result in different swapping rules. Messy but doable, and will run in $\Theta(n)$.
- Another approach is to create three temporary arrays (call them B, C, D) of the same size as the part that is being partitioned (end-start). Then simply iterate over the elements in A and copy them to either B, C , or D depending on the two comparisons. Finally, at the end, you need to copy B into A, then C into A right after B, and D right after C. This also takes $\Theta(n)$.

A common mistake was to assert that it only takes $2n$ comparisons to figure out where elements belong. While this is true, this is not constructive (nor does it necessarily even argue a lower bound). Some people claimed $2n$ comparison, plus insertion into the right place. This kind of approach yields a $\Theta(n^2)$ solution, as an insertion into an array takes $\Theta(n)$ unless it's being inserted at the end.

(c) [**11 points**] Analyze the worst-case running time of the algorithm.

Solution: This algorithm is not randomized, so there is no need to worry about 'unlucky' splitting or any such thing. However the relationship between the two medians is significant.

First of all, due to lines $3, 4$, we have to recurse twice on two separate arrays of half the size.

Line 7 contributes $\Theta(n)$ work.

Line 9 contributes some amount of work (to be analyzed below).

This gives us a recurrence of

$$T(n) = T(n/2) + T(n/2) + \Theta(n) + \textbf{line 9}$$

The remaining work due to line 9 will have the form $T(x)$ where x denotes the size of the subarray on which we make the recursive call to SUPERSELECTION in line 9. To bound x, we consider the following:

- Suppose m and M are very close together. For example, if A is an n-element array containing the sequence $< 1, 3, 5, \ldots n-1, 2, 4, 6, \ldots n >$, then C will end up containing only 2 elements, while B and D will both end up with $n/2 - 1$ elements. Therefore, we know that x can be roughly as large as $n/2$.
- If we end up recursing on B, then we know that x will be at most $n/2$, since the $n/4$ elements greater than m from the first half of A and the $n/4$ elements greater than M from the second half of A do not belong to B.
- If we end up recursing on C, then we know that x will be at most $n/2$, since the $n/4$ elements less than m from the first half of A and the $n/4$ elements greater than M from the second half of A do not belong to C.
- If we end up recursing on D, then we know that x will be at most $n/2$, since the $n/4$ elements less than m from the first half of A and the $n/4$ elements less than M from the second half of A do not belong to D.

Therefore, we can say that $x \leq n/2$ and that this is essentially tight in the worst case. Thus,

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + \Theta(n) + T(n/2) \\ &= 3T(n/2) + \Theta(n) \\ &= \Theta(n^{\log_2 3}) \end{aligned}$$

Common mistakes included incorrectly computing the amount of work due to line 9 (the most common wrong answers were either $T(n-2)$ or $T(n/3)$), or failing to solve the recurrence correctly.

Problem 4. General matrix multiplication [**15 points**]

From Lecture 3, we know that two $n \times n$ matrices A and B can be multiplied in time $O(n^{\log_2 7})$, beating the simple cubic-time algorithm. However, what if we need to multiply two non-square matrices ? For example, what if we are given two *rectangular* matrices:

- matrix A, with n rows and m columns, and
- matrix B, with m rows and n columns

such that $m \leq n$? The naive algorithm for computing $A \times B$ would take $n^2 m$ time. Is there a better algorithm ?

Your goal is to give an algorithm that has running time $o(n^2 m)$ (i.e., is asymptotically faster than the naive algorithm) whenever $m = \omega(1)$ (i.e., m is superconstant in n). You can use Strassen's algorithm as a black box.

Your solution should follow the following outline:

(a) [**10 points**] Give an algorithm as specified above, assuming that m divides n.

Solution: Let us start from the basics: in the product $A \times B$, the $(i, j)^{th}$ element is obtained by taking a dot product of the i-th *row* of A with the j-th *column* of B. Therefore, the product is an $n \times n$ matrix (not $m \times m$). This mistake cost some people a few points, since the algorithm for the other case (essentially corresponding to multiplying $B \times A$) is a little bit different. Note that the naive algorithm for computing $A \times B$ takes $O(n^2 m)$ time, while $B \times A$ can be computed in $O(nm^2)$.

Once we know what we want to do, let us focus on *how* to do it more efficiently. Since we know that m divides n, let $k = n/m$. As in Lecture 3, we can view A as a matrix of k submatrices, i.e., $A = [A_1 \dots A_k]^T$, where each A_i is a square $m \times m$ matrix (the ^{T} symbol transposes the argument, so that the A_i's are stacked vertically in A). Similarly, we can represent B as $[B_1 \dots B_k]$. Finally, we can represent $C = A \times B$ as a $k \times k$ matrix of $m \times m$ matrices C_{ij}, such that $C_{ij} = A_i \times B_j$.

In order to compute C, it suffices to compute all $C'_{ij}s$. Each $C_{ij} = A_i \times B_j$ can be computed using Strassen's algorithm, in $O(m^{\log_2 7})$ time. Since there are k^2 products to perform, the total running time of the algorithm is

$$O(k^2 m^{\log_2 7}) = O((n/m)^2 m^{\log_2 7})) = O(n^2 m^{\log_2 7 - 2})$$

Since $\log_2 7 - 2 = 0.81... < 1$, the algorithm is asymptotically faster than the naive one as long as m is not a constant (i.e., grows to infinity as n tends to infinity).

Comments: Several alternative solutions were proposed, e.g.,:

- Pad the input matrices with additional rows and columns (with entries equal to 0) to make them $n \times n$ matrices, and then apply Strassen's algorithm. This gives $O(n^{\log_2 7})$ time, which beats $O(n^2m)$ for large enough m's. It is a correct solution, but clearly quite suboptimal one: for small m's the algorithm runs slower than the naive one.

- Apply Strassen's recursive approach to non-square matrices. I.e., represent A and B as 2×2 non-square matrices, and apply the same recursive algorithm using 7 recursive calls. This approach does indeed work. However, it is *not at all* obvious. The reason: Strassen's 7 recursive multiplications are not performed directly on the submatrices of A and B, but on results of addition of a few of such submatrices. In principle, this could involve adding a submatrix of A to a submatrix of B, which makes perfect sense for square matrices, but is non-sensical for rectangular ones, since the dimensions of submatrices of A and B are reversed. It happened not to be the case (Strassen's algorithm only adds together submatrices of the same matrix). But this should be verified in order to obtain full points. Without the verification, the algorithm is very much incomplete (its correctness is suspect).

(b) **[5 points]** Show that one can assume (without loss of generality) that m divides n.

Solution: The simplest solution is to do the following. Let $n = km + l$, $0 < l < m$. Add $m - l$ rows to A (forming A') and $m - l$ columns to B (forming B'). It *does not* matter what the entries in those rows are. Since the dimensions of A and B satisfy the condition from part (a), we can now use our earlier algorithm to compute $C' = A' \times B'$. It remains to observe that the each entry $c'_{ij}, i, j = 1 \ldots n$, is equal to the (i, j)-th entry of $A \times B$. In other words, $A \times B$ is an upper left submatrix of $A' \times B'$. So, we can get get the solution from the matrix C'.

Since $n + (m - l) \leq 2n$, the time needed to compute $A' \times B'$ is still the same as before, up to constant factors (this ought to be said in order to obtain full points).

Comments: People were quite "creative" in solving this part of the problem. This was most likely due to the fact that this was the last sub-problem in the quiz, so the solutions were written under heavy influence of adrenaline.

Here are some alternative solutions (or "solutions"):

- Take A'' to be a submatrix consisting of the first km rows of A; define B'' in the analogous way. Multiply A'' by B'' using the earlier algorithm. Now we have to deal with the remainder $\tilde{A}$ of A and remainder $\tilde{B}$ of B. To complete the multiplication, we need to compute $\tilde{A} \times B$, $A \times \tilde{B}$ and $\tilde{A} \times \tilde{B}$. The ways to achieve this goal were numerous and creative:
  - Sweep it under the rug, i.e., claim this can be done using naive algorithm, and the resulting additional cost is negligible. Not true: if $n = 2m - 1$, then $l = m - 1$, and thus the remainder matrices have almost the same size as the input ones. In that case, the running time is $\Omega(m^3) = \Omega(n^3)$.

- Perform the multiplications recursively. This works, but solving the resulting recurrence is pretty non-trivial. Indeed, the most popular method for solving it was ... intimidation ("clearly, this solves to ..."). Intimidation did not work, and points were subtracted.
- Forget about some multiplications. The nice feature of this approach is that it simplifies the recursive equation considerably (some terms disappear). The bad feature is that, well, it is not correct.
- Write something confusing, related to the problem in general terms. The creativity of this approach was lost on the grading team.