Midterm Exam Solutions

Math 182, Fall 2021

Instructions: Answer each question in the space provided. If you run out of room, use the blank pages at the end. You may consult a single, double-sided page of notes. You may not use a calculator, phone or computer. If you believe there is a typo in a question, you may raise your hand to ask about it.

If you cannot figure out how to solve a problem, you may receive partial credit for solving an easier version of the same problem, as long as you state clearly that that's what you are doing.

Advice: The exam consists of five problems, plus one extra credit problem. Note that the last two problems are worth more points than the others, so if you are stuck on a problem early in the exam, it may be worth setting it aside for a while to work on the later problems.

Name: ______________________________

UID: ______________________________

Question	Points	Score
1	6	
2	9	
3	8	
4	15	
5	12	
6	0	
Total:	50	

Don't turn over this page until you are told to do so.

Question 1: Big-O and Friends (6 points)

State whether each of the following is true or false. If true, briefly explain why. If false, give a counterexample. You do not need to prove that your answers are correct.

(a) If $f, g, h, k \colon \mathbb{N} \to \mathbb{N}$ are functions such that $f \in \Theta(g)$, $h \in \Theta(k)$, and for all $n \in \mathbb{N}$, $f(n) > h(n)$ and $g(n) > k(n)$ then $f(n) - h(n) \in O(g(n) - k(n))$.

Solution: $\boxed{\text{False.}}$

Here's a counterexample. Let $f(n) = n^2 + n$, $g(n) = n^2 + 1$, and $h(n) = k(n) = n^2$. These functions meet the conditions stated in the question, but $f(n) = h(n) = n$ and $g(n) - k(n) = 1$ and $n \notin O(1)$.

Common Mistakes: Almost everyone got this one wrong. The most common mistake for people who gave proofs was failing to be sufficiently careful with manipulating inequalities in the presence of negative numbers. For example, suppose $n \in \mathbb{N}$ and C_1 and C_2 are constants such that $f(n) \leq C_1 g(n)$ and $h(n) \geq C_2 k(n)$. Then we have

$$f(n) - h(n) \leq C_1 g(n) - C_2 k(n)$$

but we *cannot* conclude that

$$f(n) - h(n) \leq \max(C_1, C_2)(g(n) - k(n))$$

The reason is that if $C_1 > C_2$ then by replacing C_2 with C_1 on the right hand side, we actually make the right hand side *smaller* not *larger* and so the inequality may no longer hold. This is because $-k(n)$ is negative.

If you're still not sure what's wrong with your proof, try tracing through what it says when f, g, h and k are as in the counterexample above.

(b) For any functions $f, g \colon \mathbb{N} \to \mathbb{N}$, $\max(f(n), g(n)) \in \Theta(f(n) + g(n))$.

Solution: $\boxed{\text{True.}}$

For any nonnegative functions f and g,

$$\max(f(n), g(n)) \leq f(n) + g(n)$$

and

$$f(n) + g(n) \leq 2\max(f(n), g(n)).$$

and therefore $\max(f(n), g(n)) \in \Theta(f(n) + g(n))$.

Common Mistakes: Several people gave counterexamples in which either f or g take on negative values. These counterexamples are not valid for this problem because the problem states that f and g are functions from $\mathbb{N}$ to $\mathbb{N}$ and $\mathbb{N}$ consists of only nonnegative integers. However, these examples would work if the problem statement had not included that condition.

Such answers would have received partial credit if they included an explicit statement that they were altering the problem statement to make it easier to solve. However, nobody did this.

Common Mistakes: Another common error was to give the correct answer but with an incomplete or excessively vague explanation. For example, stating that "the higher order term dominates" was not considered a sufficient explanation because it is not clear in this context what "higher order term" means (in class we only talked about "higher order terms" when $f(n)$ and $g(n)$ were polynomials, not arbitrary functions).

Likewise, explanations which assumed that either $f(n)$ is always larger than $g(n)$ or $g(n)$ is always larger than $f(n)$ were not considered complete since this is not always the case.

Question 2: Find the Running Time (9 points)

Find the running time of each of the algorithms below. You should show your work, but you do not need to prove that your answers are correct.

(a) This algorithm does nothing useful.

```
Foo(n):
  if n > 10:
    Foo(ceiling(n/3))
    Foo(ceiling(n/3))
    Foo(ceiling(n/3))
```

Solution: $\boxed{\text{The running time is } \Theta(n).}$

Let $T(n)$ be the running time of this algorithm. Then we have the following recursive formula for $T(n)$:

$$T(n) = 3T(\lceil n/3 \rceil) + \Theta(1).$$

Since $\log_3(3) = 1$ and $1 > 0$, the master theorem tells us that $T(n) \in \Theta(n^1) = \Theta(n)$.

(b) This algorithm removes all negative numbers from an array by going through the entries in the array one-by-one. When it encounters a negative number, it removes it from the array and moves everything after it down by one position.

```
RemoveNegatives(A):
  i = 1
  while i $\leq$ length(A):
    if A[i] < 0:
      Delete(A, i)
    else:
      i = i + 1

Delete(A, i):
  for j = i, i + 1, ..., length(A) - 1:
    A[i] = A[i + 1]
  decrease length(A) by one
```

Solution: $\boxed{\text{The running time is } \Theta(n^2).}$

In the worst case, `Delete` is called for each element of `A` (for example if every element is negative). Since the running time of `Delete(A, i)` is proportional to the length of A minus i, this means that in the worst case, the running time of the entire algorithm is proportional to

$$\sum_{i=1}^{n} (n - i) = \sum_{j=0}^{n-1} j.$$

This is just the triangular sum from 0 to $n - 1$ and we have seen in class that it is in $\Theta(n^2)$.

Common Mistakes: The most common mistake for this problem was to assert that `Delete` takes $\Theta(n)$ time. This is not exactly true since it depends on what input `Delete` is called on. In this case, making that assumption gives the right answer but in some cases it will lead you astray (for example, when an algorithm has some subroutine that is costly in the worst case scenario but which the algorithm somehow ensures is almost never called with worst case inputs).

The lesson here is that even if you only care about the worst case running time of algorithms, you cannot assume that subroutines always run as slowly as their worst case running time; you have to actually consider what inputs the algorithm is calling them with and do a more careful analysis.

(c) This algorithm is a modification of mergesort. To sort a list it first recursively sorts the left and right halves of the list. To merge the two sorted halves together, it concatenates them and then calls `Mergesort` on the resulting list.

```
SillySort(A):
  n = length(A)
  if n = 0 or n = 1:
    return A
  left = SillySort(A[1...floor(n/2)])
  right = SillySort(A[floor(n/2) + 1...n])
  B = new list whose fist half is left and whose second half is right
  return Mergesort(B)
```

Solution: $\boxed{\text{The running time is } \Theta(n\log^2(n)) = \Theta(n\log(n)\log(n)).}$

On a list of length n, `SillySort` does $\Theta(n\log(n))$ work ($\Theta(n)$ time to split the list into two parts and then create a new list from the two sorted sublists, and then $\Theta(n\log(n))$ time to run `Mergesort`) and makes two recursive calls on lists of length at most $\lceil n/2\rceil$. By drawing out the recursive tree, we can see that the total running time is thus

$$\begin{aligned}
\sum_{i=0}^{\log(n)} 2^i((n/2^i)\log(n/2^i)) &= \sum_{i=0}^{\log(n)} n(\log(n) - i)\\
&= \sum_{i=0}^{\log(n)} n\log(n) - ni\\
&= n\log^2(n) - n\sum_{i=0}^{\log(n)} i\\
&= n\log^2(n) - n\frac{\log(n)(\log(n)+1)}{2}\\
&= \Theta(n\log^2(n)).
\end{aligned}$$

Comment: This was one of the trickiest problems on the exam. My intention was that parts (a) and (b) of this problem would test your ability to recognize common patterns for running times of algorithms and that this problem would go a little deeper and require you to think about how to find the running times of algorithms that don't match the most common patterns. Because of that, this problem was more challenging.

That also means this problem may have taken more time to solve. As a general piece of advice on exam taking, if a problem is taking you a while to solve but is not worth many points, it is often a good idea to set it aside and work on the rest of the exam for a while, only returning to it if you have time at the end (or are also stuck on all the other remaining problems).

Common Mistakes: Many people tried to use the master theorem for this problem. However, the master theorem only applies for recurrences of the form $T(n) = aT(\lceil n/b \rceil) + \Theta(n^d)$ where a, b, and d are nonnegative integers, whereas the recurrence relation for the running time of this algorithm has the form $T(n) = aT(\lceil n/b \rceil) + \Theta(n \log(n))$.

Common Mistakes: Another common mistake was to ignore the fact that `Mergesort` takes $\Theta(n \log(n))$ time. If it only took $\Theta(n)$ time then the master theorem would apply and would tell us that the running time of `SillySort` is $\Theta(n \log(n))$.

Question 3: Greedy Prefix Sums (8 points)

Given an array `A`, say that a *prefix sum* of `A` is any number of the form `A[1] + A[2] + . . . + A[i]` for some i between 1 and the length of A. Suppose you are trying to solve the following problem: given a list of integers $x_1, x_2, \dots, x_n$ and a natural number $m > 1$, check whether it is possible to put $x_1, \dots, x_n$ into an array in some order such that no prefix sum of the array is divisible by m.

Example. If the list is $1, 2, 3, 4$ and $m = 3$ then it is possible since the prefix sums of `[1, 3, 4, 2]` are $1, 4, 8, 10$ and none of these are divisible by 3. If the list is $1, 2, 3, 4$ and $m = 2$ then it is not possible since the sum of all the numbers is divisible by 2.

You come up with the following greedy algorithm to solve this problem. Start with an empty array. At each step, look for a number which is still in the list and which, when added to the array built so far, does not make the sum divisible by m. If such a number exists then remove the first such number from the list and add it to the end of the array. If no such number exists, then declare that there is no solution. In pseudocode:

```
Check(X, m):
  A = new empty array
  while X is not empty:
    if there is x in X such that sum(A) + x is not divisible by m:
      remove the first such x from X and add it to the end of A
    else:
      return False
  return True
```

Is this algorithm correct? If so, prove that it is correct. If not, give an example of an input on which it is incorrect.

Solution: $\boxed{\text{The algorithm is not correct.}}$

One counterexample is the list $1, 2, 2$ with $m = 3$. The greedy algorithm will first take 1 and then get stuck and declare that there is no solution. However, there actually is a solution: `[2, 2, 1]`.

Common Mistakes: About half the class got this problem right. Many people offered incorrect proofs that the algorithm was correct. Many of these proofs fell prey to the same fairly subtle flaw. Let me try to explain what that was.

Suppose that the algorithm has run for i steps so far and we are now trying to add one more element to A. It is natural to think that if there is any way to find a valid array with $i+1$ elements from the list then there is a way to do so which extends the length i array that was found on step i. But this is not necessarily so, as the counterexample in the solution above demonstrates.

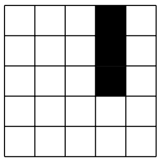
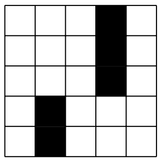
This is an instance of a kind of tricky point about proving greedy algorithms correct by induction. It is tempting to assume that if you have an optimal solution for the first i elements of some list then you can find an optimal solution for the first $i + 1$ elements simply by making a simple modification to the optimal solution solution for the first i. However, this is very often not the case and even when it is the case, showing it is often pretty hard and is the core of the proof that the greedy algorithm is correct.

Question 4: How Many Paths? (15 points)

A *path* through an $n \times n$ grid is a sequence of squares in the grid such that each square is either horizontally or vertically adjacent to the previous square. A path is called *increasing* if each square is either to the right of or above the previous square in the path.

Suppose you are given an $n \times n$ grid where some squares have been marked as blocked. You want to know many increasing paths there are in the grid which start in the bottom left corner, end in the upper right corner and do not contain any blocked squares.

Example. Two grids are shown below, with blocked squares filled in with black. In the first grid, there are exactly 5 increasing paths from the bottom left corner to the upper right corner. In the second grid, there are none.

Assume that the input comes in the following format: `A` is an $n \times n$ array such that `A[i, j]` records whether the square in the i^{th} row and j^{th} column of the grid is blocked. In particular, assume `A[i, j]` is 0 if the square is blocked and 1 otherwise. Also assume that $(1, 1)$ corresponds to the square in the bottom left of the grid, so increasing i and j corresponds to going up and to the right in the grid, respectively.

We will use dynamic programming to solve this problem. Define $P(i, j)$ to be the number of increasing paths which start at square $(1, 1)$, end at square (i, j) and do not contain any blocked squares. For the questions below, you do not need to prove your answers are correct nor do you need to show any work.

(a) Write a recursive formula (i.e. update rule) for $P(i, j)$.

Solution: There were a few valid ways to do this one. Here's one possibility.

$$P(i, j) = A[i, j] \cdot (P(i-1, j) + P(i, j-1))$$

Another way to express this same formula would be to write

$$P(i, j) = \begin{cases} P(i-1, j) + P(i, j-1) & \text{if } A[i, j] = 1 \\ 0 & \text{if } A[i, j] = 0. \end{cases}$$

The idea is that a path from $(1, 1)$ to (i, j) has to either go through square $(i-1, j)$ (i.e. it reaches square (i, j) by going to the right) or through square $(i, j-1)$ (i.e. it reaches square (i, j) by going up) but cannot go through both. Therefore, the number of paths to (i, j) is the sum of the number of paths to $(i-1, j)$ and to $(i, j-1)$, unless (i, j) is blocked, in which case the number of paths is 0.

Comment: Parts (a) and (b) of this problem were graded together because whether or not the recursive formula was correct could depend on what the base cases were. Thus for a discussion of common mistakes, look at the comments after the solution to part (b).

(b) Give the base case(s) of the recursion.

Solution:

$$\begin{aligned} P(1,1) &= A[1,1] & \\ P(i,0) &= 0 & \text{for all } 0 \le i \le n \\ P(0,j) &= 0 & \text{for all } 0 \le j \le n \end{aligned}$$

There is exactly one path from $(1,1)$ to $(1,1)$ unless $(1,1)$ is blocked, in which case there are 0 paths. Also, there are no paths to $(0,j)$ or $(i,0)$ since those do not correspond to valid squares in the grid. We could also replace the second and third lines above with the following base cases.

$$\begin{aligned} P(i,1) &= A[i,1] \cdot P(i-1,1) \\ P(1,j) &= A[1,j] \cdot P(1,j-1). \end{aligned}$$

Common Mistakes: There were a few common mistakes for this problem. In order of most to least serious, they were as follows.

- A number of solutions gave incorrect recursive formulas either involving taking the maximum of $P(i-1,j)$ and $P(i,j-1)$ instead of adding them together or adding 1 to each of them instead of adding them directly.
- Some solutions failed to account for blocked squares at all. For example, several people gave the recursive formula as $P(i,j) = P(i-1,j) + P(i,j-1)$. This counts the total number of increasing paths, not the number of increasing paths which avoid blocked squares.
- Some solutions failed to account for squares on the left and bottom edges of the grid. To see what I mean, consider trying to apply the recursive formula when either $i = 1$ or $j = 1$. In such cases, the recursive formula will include a reference to either $P(0,j)$ or $P(i,0)$, which are not defined (unless an appropriate base case is provided).
- Some solutions failed to provide a base case for $P(1,1)$.
- A common alternate recursive formula was

  $$P(i,j) = A[i-1,j] \cdot P(i-1,j) + A[i,j-1] \cdot P(i,j-1).$$

  This works in most cases, but fails if square (n,n) is blocked (to see why, try considering what this recursive formula will say on a 2×2 grid with only square $(2,2)$ blocked).
- Many solutions tried using $P(1,1) = 1$ as a base case. This is correct in most cases, but if square $(1,1)$ is blocked, it is not correct (since in that case there are no paths).

(c) What is the running time of the resulting dynamic programming algorithm?

Solution: There are $(n+1)^2$ total subproblems (including base cases) and each one can be solved in $O(1)$ time, so the total running time is $O(n^2)$.

Question 5: Search a Jumbled List (12 points)

You have decided to open up a bookstore. To make sure the books stay organized, every book is tagged with a unique integer ID and you keep the books in sorted order according to their ID. However, sometimes customers will pull a book off the shelf and put it back in a different location than they found it in. Fortunately, they always put it close to where they originally found it and no book is ever more than 2 spots away from its proper location. You would like a way to check whether a book with a given ID is contained in your store.

Design an efficient algorithm to solve this problem. Assume you are given an array of integers, L, corresponding to the ID numbers of the books in your store, and a number a, corresponding to the ID of the book you are trying to find. You may assume that no number in L is more than 2 positions away from where it would be if L was sorted in increasing order.

(a) Describe the main idea of your algorithm. You do not need to prove that your idea is correct.

Solution: There are a few different correct ways to do this, as well as a few that look correct but don't quite work. Here are three correct solutions.

Solution 1: Do binary search, but after comparing a to the middle element of the list, recurse on either the left $\lfloor n/2 \rfloor + 4$ elements or the right $\lceil n/2 \rceil + 4$ elements. Also, if there are at most 9 elements in the list, just compare each one to a instead of recursing.

Solution 2: Do binary search, but before recursing on either the left or right half of the list, compare each of the middle 9 elements to a. Also, if there at most 9 elements in the list, just compare each one to a instead of recursing.

Solution 3: Do binary search, but when it finishes, search the 9 elements in the list closest to the position where it ended.

Each of these solutions depends on the observation that if two elements of the list are out of order with respect to each other then they can be at most 4 positions apart in the list[1]. In other words, if $i < j$ and `L[i] > L[j]` (i.e. if (i, j) is an *inversion*) then $j \leq i + 4$ because each of `L[i]` and `L[j]` can be at most 2 positions away from where they would be if L were sorted and if L were sorted, `L[i]` would have a higher index than `L[j]`.

Solutions 1 and 2 are the easiest to prove correct: prove by induction that at each step, if a is in the list at all then it is within the current interval being considered. To prove solution 3 is correct, prove that if a is in the list at all then it is either within the current interval or no more than 4 positions away from one of the endpoints of the current interval.

Common Mistakes: Many answers to this question seemed to be based on a misunderstanding of the question. For example, some solutions attempted to sort the list. Other solutions seemed to assume that a was an index in the list rather than an item that we are trying to find in the list.

Common Mistakes: Several people gave correct solutions that were much slower than necessary. For example, some solutions began by sorting the list. However, sorting a list takes $\Theta(n \log(n))$ time (at least if you just use a generic sorting algorithm that does not make any special assumptions about the list) which is already slower than just scanning through the entire list and comparing each element to a (which takes $\Theta(n)$ time).

[1]Actually, they can be at most 3 positions apart, but that's a bit less obvious.

Common Mistakes: Among solutions that were close to correct, the most common mistake was to use one of the solutions above, but search too small of a neighborhood (e.g. only look at the middle 5 elements rather than the middle 9).

Another somewhat subtle mistake was to only recurse on the left half of the list if a is smaller than all of the 5 middle elements of the list and likewise only recurse on the right half of the list when a is larger than all of the 5 middle elements. The idea, presumably, is that the true median element of the list is always among these 5 elements. That much is true, but this algorithm does not always work correctly in the case where a is neither smaller nor larger than all 5 middle elements. The point is that this *does not* always mean that a is not in the list, since while the middle 5 elements do contain the true median, they also contain other elements that could be as many as 4 positions away (on either side) from the median if the list were sorted.

Many solutions also failed to provide a valid base case. This was mostly a problem for solutions that were similar to solutions 1 and 2 above. For such solutions, the recursive step does not make sense when the size of the interval is smaller than 9. The easiest thing to do at that point is just to search through the elements of the list one-by-one. This does not make the asymptotic running time worse since it adds at most a constant number of operations to the running time (since we know the size of the interval must be small in that case).

(b) Show pseudocode to implement your algorithm. Note that if you run out of time to give pseudocode but the idea you described in part (a) is correct, you will still receive most of the credit for this problem.

Solution: Below are implementations of each of the three algorithms described above. For each one, the full list can be searched by running `Find(L, a, 1, length(L))`.

Solution 1:

```
Find(L, a, i, j):
  if i + 9 > j:
    for k = i, i + 1, ..., j:
      if L[k] = a:
        return True
    return False
  mid = floor((i + j)/2)
  if L[mid] = a:
    return True
  if L[mid] < a:
    return Find(L, a, mid - 4, j)
  else:
    return Find(L, a, i, mid + 4)
```

Solution 2:

```
Find(L, a, i, j):
  if i + 9 > j:
    for k = i, i + 1, ..., j:
      if L[k] = a:
        return True
    return False
  mid = floor((i + j)/2)
  if L[mid] = a:
    return True
  for k = mid - 4, ..., mid + 4:
    if L[k] = a
      return True
  if L[mid] < a:
    return Find(L, a, mid + 1, j)
  else:
    return Find(L, a, i, mid - 1)
```

Solution 3:

```
Find(L, a, i, j):
  if i > j:
    for k = min(j - 4, 1), ..., max(i + 4, length(L)):
      if L[k] = a:
        return True
    return False
  mid = floor((i + j)/2)
  if L[mid] = a:
    return True
  if L[mid] < a:
    return Find(L, a, mid + 1, j)
  else:
    return Find(L, a, i, mid - 1)
```

(c) What is the running time of your algorithm? You do not need to show any work.

Solution: Each of the three algorithms has a worst case running time of $\Theta(\log(n))$, where n is the length of L. This is because each one cuts the list (almost) in half on each step until reaching a base case which takes $O(1)$ time.

Comment: The answer above is correct, but it obscures some differences between the running times of the three algorithms. Consider the generalization of the problem where each element of the list is at most k positions away from where it would be if the list were sorted. Then solutions 1 and 3 both take $\Theta(\log(n)+k)$ time, but solution 2 takes $\Omega(\min(n, k\log(n)))$ time. For large values of k, this is much worse. This is relevant to the extra credit question.

Question 6: Extra Credit (1 point (bonus))

This problem is optional. If you find a correct solution, you will receive one point of extra credit.

Suppose that in the previous problem, instead of every book being at most 2 spots away from its correct position, every book is at most $\log(n)$ spots away, where n is the total number of books. Find an $O(\log(n))$ time algorithm to solve this version of the problem. To receive credit for this problem you just need to describe the main idea of your algorithm; you do not need to give pseudocode or prove that your algorithm is correct.

Solution: Solutions 1 and 3 from problem 5 can both be easily modified to solve this problem—essentially just replace 2 with $\log(n)$ everywhere in the algorithm (more precisely, replace 4 and 9 with $2\log(n)$ and $4\log(n)+1$, respectively). The reason this gives an $O(\log(n))$ algorithm for both solutions 1 and 3 is that each one does a constant amount of work on each recursive step and takes $\Theta(\log(n))$ steps to reach the base case, where they then do $O(\log(n))$ work.

Solution 2, however, is too slow to work for this problem. The issue is that if it is modified to change 2 to $\log(n)$ then it will do $\Theta(\log(n))$ work on each recursive step, for a total running time of $\Theta(\log^2(n))$.