Final

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on the top of every page of this quiz booklet.
- You have 180 minutes to earn a maximum of 180 points. Do not spend too much time on any one problem. **Read through all problems first**, then solve them in an order that allows you to make the most progress.
- **You are allowed three double-sided letter-sized sheet with your own notes**. No calculators, cell phones, or other programmable or communication devices are permitted.
- Write your solutions in the space provided. Pages will be scanned and separated for grading. If you need more space, write "Continued on S1" (or S2, S3, S4, S5, S6) and continue your solution on the referenced scratch page at the end of the exam.
- Do not waste time and paper rederiving facts that we have studied in lecture, recitation, or problem sets. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required. Be sure to **briefly** argue that your **algorithm is correct**, and analyze the **asymptotic running time of your algorithm**. Even if your algorithm does not meet a requested bound, you **may** receive partial credit for inefficient solutions that are correct.

Problem	Parts	Points
0: Information	2	2
1: Decision Problems	10	40
2: Debugging	3	9
3: Shelf Sort	3	12
4: 2D Priorities	1	16
5: Merged Anagrams	1	16
6: Positive Unicycle	1	15
7: Teleportation	1	15
8: Dinner Diplomacy	1	15
9: Sweet Tapas	1	20
10: Gokemon Po	1	20
Total		180

Name: ______________________________

School Email: ______________________________

Problem 0. [2 points] **Information** (2 parts)

(a) [1 point] Write your name and email address on the cover page.

Solution: OK!

(b) [1 point] Write your name at the top of each page.

Solution: OK!

Problem 1. [40 points] **Decision Problems** (10 parts)

For each of the following questions, circle either **T** (True) or **F** (False), and **briefly** justify your answer in the box provided (a single sentence or picture should be sufficient). Each problem is worth 4 points: 2 points for your answer and 2 points for your justification. **If you leave both answer and justification blank, you will receive 1 point**.

(a) T F If $f(c) = O(1)$ for a constant c and $f(n) = 4f(n/2) + \Theta(n^2)$ for $n > c$, and $g(c) = O(1)$ for a constant c and $g(n) = 8g(n/2) + \Theta(n)$ for $n > c$, then $f = \Omega(g)$.

Solution: False. f is $\Theta(n^2 \log n)$ and g is $\Theta(n^3)$, by Master theorem. So $f \in o(g)$ and $f \notin \Omega(g)$, since $\Omega(g) \cap o(g) = \emptyset$.

(b) T F Given an array of n integers representing a binary min-heap, one can find and extract the maximum integer in the array in $O(\log n)$ time.

Solution: False. The maximum element could be in any leaf of the heap, and a binary heap on n nodes contains at least $\Omega(n)$ leaves.

(c) T F Any binary search tree on n nodes can be transformed into an AVL tree using $O(\log n)$ rotations.

Solution: False. Since any rotation changes height of any node by at most a constant, a chain of n nodes would require at least $\Omega(n - \log n)$ rotations.

(d) T F Given an array of n key-value pairs, one can construct a hash table mapping keys to values in **expected** $O(n)$ time by inserting normally into the hash table one at a time, where collisions are resolved via chaining. If the pairs have unique keys, it is possible to construct the same hash table in **worst-case** $O(n)$ time.

Solution: True. Because keys are unique, we do not have to check for duplicate keys in chain during insertion, so each insert can take worst-case $O(1)$ time.

(e) T F You implement a hash table using some specific hash function $h(k)$, and then insert a sequence of n integers into the hash table. Then looking up whether a queried integer is stored in this hash table will take expected $O(1)$ time.

Solution: False. Expected bounds are only valid when choosing a hash function randomly from a universal hash family.

(f) T F Given a graph where all edge weights are strictly greater than -3, a shortest path between vertices s and t can be found by adding 3 to the weight of each edge and running Dijkstra's algorithm from s.

Solution: False. Counter example: a graph on vertices s, t, v, with undirected weighted edges $w(s, v) = 0$, $w(s, t) = -1$, and $w(v, t) = -2$.

(g) **T F** Given a directed acyclic graph having positive edge weights and only one source vertex s having no incoming edges, running Dijkstra's from s will remove vertices from the priority queue in a topological sort order.

Solution: False. Counter example: a graph on vertices s, t, v, with directed weighted edges $w(s,t) = 1$, $w(s,v) = 2$, and $w(v,t) = 1$. Dijkstra from s processes vertices in order (s,t,v), while topological sort order is (s,v,t).

(h) **T F** Given an unweighted directed graph having only one source vertex s having no incoming edges, a depth-first search from s will always generate a DFS tree containing a longest simple path in the graph.

Solution: False. For a longest simple path to exist in the DFS tree, DFS would need to search edges in the longest path before others, while a valid DFS may chose to first search any one of its unvisited neighbors.

(i) **T F** The problem of determining whether an undirected graph contains as a subgraph a complete graph on k vertices is in NP.

Solution: True. Providing a subset of k vertices is a $O(k)$ polynomial size certificate which can be checked for completeness in $O(k^2)$ polynomial time.

(j) **T F** If there is a polynomial time algorithm to determine whether an unweighted graph contains a Hamiltonian path, then there is also a polynomial time algorithm to determine whether a weighted graph contains a simple path with weight greater than or equal to k.

Solution: True. Both problems are NP-Hard and in NP, so if you can solve one of them in polynomial time, you can solve any problem in NP in polynomial time, including the other problem.

Problem 2. [9 points] **Debugging** (3 parts)

Given a binary max-heap stored in Python list `A`, the following code is supposed to extract and return the maximum element from the heap while maintaining the max-heap property, but does not always work; the code contains a bug.

```
 1  def extract_max(A):
 2      # Given array A satisfying the max-heap property,
 3      # extracts and returns the maximum element from A,
 4      # maintaining the max-heap property on the remainder of A
 5      i, n = 0, len(A)
 6      A[0], A[n - 1] = A[n - 1], A[0]                 # swap first and last
 7      while True:
 8          l = (2 * i + 1) if (2 * i + 1 < n) else i   # left child or i
 9          r = (2 * i + 2) if (2 * i + 2 < n) else i   # right child or i
10          c = l if A[r] > A[l] else r                 # larger child
11          if A[i] >= A[c]:                            # satisfies max-heap!
12              break
13          A[i], A[c] = A[c], A[i]                     # swap down
14          i = c
15      return A.pop()                                  # return maximum
```

(a) [2 points] State the running time of the code above.

Solution: Amortized $O(\log n)$ time. Amortized not necessary for full credit.

Rubric:

- 2 points for $O(\log n)$

(b) [3 points] Fix the bug in the code by changing a single line.

Solution: Change `A[r] > A[l]` in line 10 to `A[r] < A[l]`. In fact there is another bug in the code: `A.pop()` should occur prior to the `while` loop, storing the maximum value to return later. Students received full points for mentioning either bug.

Rubric:

- 3 points for a correct line change

(c) [4 points] Give an array A of four integers satisfying the max-heap property for which the original code does not behave as desired.

Solution: $A = [3, 1, 2, 0]$ would yield array $[1, 0, 2]$, which does not satisfy the max-heap property. Any max-heap on four distinct values would suffice.

Rubric:

- 4 points for a valid max-heap on four distinct values

Problem 3. [12 points] **Shelf Sort** (3 parts)

You have n books in your room, and you want to store them on a single shelf. Each book has the same thickness. **Briefly** describe and justify an algorithm that is **best suited** to the following scenarios (in one or two sentences).

(a) [4 points] You want to store your books on a single shelf sorted by color. The color of each book is specified by tuple of red, green, and blue values, each in the range 1 to 16. Place your books on the shelf sorted first by red, then green, then blue value.

Solution: Radix sort the books by color, using counting sort to sort first by blue value, placing the books into at most 16 stacks, then sort by green, then red value. This is very efficient as each book is moved four times.

(b) [4 points] You are sick of the colorful ordering of your books, and instead want to sort your books by height. The only way to compare the height of two books is to take them off the shelf and compare them. You would like to sort your books by only removing two books from the shelf at a time, then putting them back in the same or swapped order. Sort your books by height using swaps.

Solution: Use heap sort to sort the books using at most $O(n \log n)$ swaps. We choose heap sort because it is an optimal comparison sort algorithm that is also in place.

(c) [4 points] You've bought three new books and want to add them to your bookshelf currently sorted by height, but you only want to move any book currently on the shelf at most once. Place the new books on the shelf, assuming you may take a book off the shelf, compare it with a new book, and place it back on the shelf wherever you like.

Solution: Insertion sort the three books down from the right to their respective places, by removing each book, comparing to the largest unplaced new book, and placing it back on the shelf to right of its previous location by the number of new books having smaller height. This algorithm requires at most n moves and at most n comparisons.

Rubric:

- 1 point for a correct algorithm
- 2 points for algorithm that is well suited to problem
- 1 point for justification

Problem 4. [16 points] **2D Priorities** (1 part)

A **2D priority queue** maintains a set of ordered integer pairs (x, y), and supports four operations:

- `insert(x,y)`: insert ordered pair (x, y) into the queue
- `extract(x,y)`: remove ordered pair (x, y) from the queue, or return `None` if queue is empty
- `extract_min_x()`: remove and return an ordered pair from the queue having smallest x coordinate, or return `None` if queue is empty
- `extract_min_y()`: remove and return an ordered pair from the queue having smallest y coordinate, or return `None` if queue is empty.

Describe a data structure that supports all four 2D priority queue operations, each in **worst-case** $O(\log n)$ time, where n is the number of pairs in the queue at the time of the operation. You may assume that each pair stored in the queue is unique.

Solution: Here are two possible solutions using AVL trees. All AVL operations are worst-case.

1) Store each pair in two AVL trees, tree X sorted lexicographically by x then y, and tree Y sorted lexicographically by y then x. For `insert(x,y)`, insert into each AVL tree in $O(\log n)$ time. For `extract(x,y)`, find and remove the pair in each AVL tree, also in $O(\log n)$ time. For `extract_min_x()`, find the minimum pair (x', y') from pairs in AVL tree X, where x' is minimum, and extract from X in $O(\log n)$ time. Then find (x', y') in AVL tree Y and extract from Y in $O(\log n)$ time. The procedure for `extract_min_y()` is symmetric.

2) Store pairs in a single AVL tree sorted lexicographically by x then y. Support `insert(x,y)` and `extract(x,y)` directly using AVL `insert` and `delete`. Augment each node n with the minimum y for any node in n's subtree, which can be maintained during dynamic operations in $O(\log n)$ time. Specifically, $n.ymin = \min(n.key.y, n.left.ymin, n.right.ymin)$. To extract a pair with minimum x, extract and return the node in the AVL with minimum key in $O(\log n)$ time, as finding minimum and deletion on an AVL tree can both be done in $O(\log n)$ time. To extract a pair with minimum y in the subtree rooted at node n, compare $n.key.y$, $n.left.ymin$, and $n.right.ymin$. If $n.key.y$ is minimum, extract and return n. Otherwise, if $n.left$ is minimum, recursively extract minimum in $n.left$'s subtree, and similarly for $n.right$. This procedure walks down the tree of height $O(\log n)$ and makes a single extraction, so can be done in $O(\log n)$ time.

Rubric:

- 2 points for using an AVL or other balanced BST
- 2 points each for correct `insert(x,y)` and `extract(x,y)`
- 3 points each for correct `extract_min_x()` and `extract_min_y()`
- 1 point for correct running time analysis for each operation
- Partial credit may be awarded

Problem 5. [16 points] **Merged Anagrams** (1 part)

A list of words contains a **merged anagram triple** (a, b, c) if words a and b from the list can be concatenated to form an anagram of another list word c. For example, the list of words (`grail`, `happy`, `word`, `test`, `moths`, `algorithms`) contains the merged anagram triple (`grail`, `moths`, `algorithms`). Given a list of n words, where each word contains at least one and at most n English characters, describe an $O(n^2)$ time algorithm to determine whether the list contains a merged anagram triple. State whether your algorithm achieves a **worst-case**, **expected**, and/or **amortized** bound.

Solution:

For each word, construct a histogram of the number of times a letter from $a = 1$ to $z = 26$ occurs in the word. Assuming that value n can be stored in a constant number of words, each of these histograms has constant size, and can be computed in $O(n + 26)$ time. Doing this for all words takes worst-case $O(n^2)$ time. The approach is to construct a dictionary on words or word pairs, then lookup collisions with pairs or single words.

1) Use hash table with expected $O(1)$ lookup. Construct hash table mapping each word's histogram to one word from the list, overwriting if multiple. Inserting each takes amortized expected $O(1)$ time, so inserting all words takes expected $O(n)$ time. Then, for each pair of distinct words (a, b), sum their histograms and lookup the combined histogram in the hash table in expected $O(1)$ time. If the combined histogram maps to some word c, then a and b's histograms combine to form an anagram of c, so return (a, b, c). Doing this for all pairs takes expected $O(n^2)$ time.

2) Use sorted array with worst-case $O(\log n)$ lookup. For each pair of words (a, b), store tuple (x, a, b) in an array, where x is the combined histogram of the histograms of a and b. This array contains $O(n^2)$ tuples, and each takes worst-case $O(1)$ time to compute. Sort this array by histogram x using radix sort, sorting the frequencies of each letter ranging from 0 to $2n$ using counting sort in $O(n + n^2)$ time, once for each letter in worst-case $O(26(n + n^2)) = O(n^2)$ time. For each word c, lookup its histogram using binary search in $O(\log n^2) = O(\log n)$ time. If there is a match (x, a, b) where x is the histogram of c, return (a, b, c). Performing this search for each word takes worst-case $O(n \log n)$ time, so the algorithm takes worst-case $O(n^2)$ time.

Rubric:

- 12 points for a correct algorithm. If using dictionary:
  - 4 points for converting words to histograms
  - 4 points for correctly building a dictionary
  - 4 points for correctly lookup in dictionary
- 2 points for correct running time analysis
- 2 points if algorithm is worst-case or expected $O(n^2)$
- Partial credit may be awarded

Problem 6. [15 points] **Positive Unicycle** (1 part)

Given source vertex s from a weighted directed graph $G = (V, E)$ having both positive and negative edge weights containing **exactly one positive weight cycle** (though possibly having many cycles of negative weight), describe an $O(|V|^3)$ algorithm to return a list of all vertices having a longest path from s traversing fewer than $|V|$ edges.

Solution: All simple longest paths will traverse fewer than $|V|$ edges, whereas longest paths which are non-simple will contain a positive weight cycle. Negate edge weights in G and run Bellman-Ford from s to determine whether the positive weight cycle is reachable from s. If not, all vertices reachable from s satisfy the condition as all longest paths traverse fewer than $|V|$ edges. Otherwise, Bellman-Ford finds all vertices reachable from s as well as finding the positive weight cycle reachable from s. Any vertex reachable from arbitrary vertex v on the cycle will not have a longest path from s traversing fewer than $|V|$ edges, so run BFS or DFS to find the set of vertices reachable from v. Then, return the set of vertices reachable from s but not reachable from v. Bellman-Ford runs in $O(|V||E|) = O(|V|^3)$ time and traversal takes $O(|V| + |E|) = O(|V|^2)$ time, so this algorithm runs in $O(|V|^3)$ time.

Rubric:

- 9 points for a correct algorithm. Common approach:
  - 3 points for algorithm to find positive weight cycle
  - 3 points for finding vertices reachable from cycle
  - 3 points for returning correct set
- 2 points for correct running time analysis
- 4 points if algorithm is $O(|V|^3)$
- Partial credit may be awarded

Problem 7. [15 points] **Teleportation** (1 part)

A **teleportation graph** is a graph where each vertex has been assigned a color: either black, red, or blue. A **teleportation path** in a teleportation graph is a sequence of vertices $v_1, \dots, v_k$ such that every adjacent pair of vertices (v_i, v_{i+1}) is either an edge of the graph, or v_i is red and v_{i+1} is blue, i.e. the path may **teleport** from any red vertex to any blue vertex. The **cost** of a teleportation path is the sum of weights of all edges traversed by the path; teleportation contributes zero cost. Given a weighted directed teleportation graph $G = (V, E)$ having only positive edge weights, where each vertex has **at most four** out-going edges, describe an $O(|V| \log |V|)$ time algorithm to find the minimum cost teleportation path from vertex s to t that teleports **exactly two times** (if such a path exists). Significant partial credit will be given for a correct $O(|V|^2)$ time algorithm.

Solution: To remember how many times a path has teleported, we duplicate G three times, where vertex v_k corresponds to visiting vertex $v \in V$ having already teleported k times, for $k \in \{0, 1, 2\}$. Add in two auxilliary nodes p_0 and p_1, and for $k \in \{0, 1\}$ and $v \in V$, add a directed edge from v_k to p_k if v is red, and add a directed edge from p_k to v_{k+1} if v is blue. This graph has $3|V|$ vertices and $3|E| + O(2|V|)$ edges. Since vertices have bounded out-degree, $|E| = O(|V|)$ and this graph can be constructed in $O(|V|)$ time. Run Dijkstra from s_0 to find a shortest path to t_2, which will correspond to a path from s to t in the original graph using exactly two teleports running through p_0 and p_1, and can be performed in $O(|V| \log |V| + |V|) = O(|V| \log |V|)$ time.

A simpler but less efficient approach would be to still copy the graph three times, but connect levels without using auxilliary nodes. For $k \in \{0, 1\}$ and for each pair of vertices u and v, connect u_k to v_{k+1} via a directed edge if u_k is red and v_{k+1} is blue with weight zero. This results in $O(|V|^2)$ edges between each level, so Dijkstra from s_0 to t_2 will instead take $O(|V| \log |V| + |V|^2) = O(|V|^2)$ time.

Rubric:

- 9 points for a correct algorithm. If graph duplication:
  - 3 points for graph duplication
  - 3 points for additional edges between levels
  - 3 points for shortest paths
- 2 points for correct running time analysis
- 4 points if algorithm is $O(|V| \log |V|)$
- Partial credit may be awarded

Problem 8. [15 points] **Dinner Diplomacy** (1 part)

Meresa Thay is organizing a state dinner for a large number of diplomats. Each diplomat has a **nemesis**: another diplomat who they despise the most. If a diplomat is served dinner after the diplomat's nemesis, the diplomat will file a complaint with the state department. A complaint from diplomat d_i will result in s_i units of scandal. Given a list containing the nemesis and amount of scandal associated with each of the n diplomats attending the dinner, describe an $O(n)$ time algorithm to determine whether there exists an order to serve dinner to the diplomats one-by-one resulting in no more than S total units of scandal. For example, if only two diplomats d_1 and d_2 attend, with $s_2 < s_1$, they would be nemeses of each other. There will be a serving order resulting in no more than S total units of scandal if $s_2 \leq S$, by serving dinner to d_1 first, then to d_2.

Solution: Construct a graph on the diplomats with directed edge from d_i to d_j if d_j is the nemesis of d_i, weighted by s_i. This graph has n vertices and n edges. We would like to find an ordering on the vertices such that we minimize the total scandal of all edges that are directed against the ordering. Each vertex in this graph has exactly one outgoing edge, so each connected component of k vertices has exactly k edges, and thus contains exactly one cycle. Each cycle guarantees that at least one diplomat on the cycle will file a complaint. Use DFS to explore the entire graph in $O(n)$ time. Every back edge found corresponds to a cycle in the graph. For each cycle, find the edge in the cycle whose weight is minimum, i.e., identify a diplomat on the cycle whose complaint will result in the minimum amount of scandal. Finding the minimum for all cycles can be found in $O(n)$ time simply by traversing each cycle, since the cycles are disjoint. Then simply return whether the sum of the minimum s_i from each cycle is no more than S. This algorithm runs in $O(n)$ time.

Rubric:

- 9 points for a correct algorithm. Common approach:
  - 3 points for relation to topological sort
  - 3 points for finding cycles in graph
  - 3 points for correctly breaking cycles
- 2 points for correct running time analysis
- 4 points if algorithm is $O(n)$
- Partial credit may be awarded

Problem 9. [20 points] **Sweet Tapas** (1 part)

Obert Ratkins is having dinner at an upscale tapas bar, where he will order many small plates which will comprise his meal. There are n plates of food on the menu, with each plate p_i having integer volume v_i, integer calories c_i, and may or may not be sweet. Obert is on a diet: he wants to eat no more than C calories during his meal, but wants to fill his stomach as much as he can. He also wants to order exactly s sweet plates, for some $s \in [1, n]$, without purchasing the same dish twice. Describe a dynamic programming algorithm to find a set of plates that maximizes the volume of food Obert can eat given his constraints.

Be sure to define a set of subproblems, relate the subproblems recursively, argue the relation is acyclic, provide base cases, construct a solution from the subproblems, and analyze running time. Correct but inefficient dynamic programs will be awarded significant partial credit.

Solution:

1. **Subproblems**
   - Let s_i be 1 if plate p_i is sweet and 0 otherwise
   - $x(i, j, k)$: largest volume of food orderable from plates p_i to p_n, using at most j calories, ordering exactly k sweet plates
2. **Relate**
   - Either we order plate p_i or not. Guess!
   - If order p_i, get v_i in volume but use c_i calories
   - If p_i is sweet, need to order one fewer sweet plate
   - $x(i, j, k) = \max \begin{cases} v_i + x(i+1, j - c_i, k - s_i)) & \text{if } c_i \leq j \text{ and } s_i \leq k \\ x(i+1, j, k) & \text{always} \end{cases}$
3. **DAG**
   - Subproblems $x(i, j, k)$ only depend on strictly larger i so acyclic
   - Solve in order of decreasing i, then j and k in any order
   - Base case: for all $j \in [0, C]$, $x(n+1, j, 0) = 0$ and $x(n+1, j, k) = -\infty$ for $k \neq 0$
4. **Evaluate**
   - Solution given by $x(1, C, s)$
5. **Analysis**
   - (# Subproblems) $\times$ (work per problem) = $O(nCs) \times O(1) = O(nCs)$

Rubric:

- 3 points for subproblem description (max 5 points total if subproblems exponential in n)
- 5 points for a correct recursive relation
- (2, 2, 2) points for (acyclic, base cases, solution)
- (2, 4) points (correct running time analysis, algorithm achieves solution running time)

Problem 10. [20 points] **Gokemon Po** (1 part)

Kash Etchum wants to play a new augmented reality game, Gokemon Po, where the goal is to catch a set of n monsters who reside at specific locations in her town. Monsters must be caught in a specified linear order: before Kash can catch monster m_i, she must have already caught all monsters m_j occurring before m_i in the order (for $j < i$). To catch monster m_i, Kash may either purchase the monster in-game for c_i dollars, or she may catch m_i for free from that monster's location. If Kash is not at the monster's location, she will have to pay a ride share service to drive her there. The **minimum possible cost** of transporting from monster m_i to monster m_j via ride sharing is denoted $s(i, j)$. Given a list of prices of in-game purchases and ride sharing trips between all pairs of monsters, describe a dynamic programming algorithm to determine the minimum amount of money Kash must spend in order to catch all n monsters, assuming that she starts at the location of monster m_1.

Be sure to define a set of subproblems, relate the subproblems recursively, argue the relation is acyclic, provide base cases, construct a solution from the subproblems, and analyze running time. Correct but inefficient dynamic programs will be awarded significant partial credit.

Solution:

1. **Subproblems**
   - $x(i, j)$: min cost of catching monsters m_i to m_n starting at location m_j for $j \leq i$
2. **Relate**
   - Either acquire monster m_i by purchasing or ride-sharing to location. Guess!
   - If purchase spend c_i dollars, else need to ride share to m_i from m_j
   - $x(i, j) = \begin{cases} \min(c_i + x(i+1, j), s(j, i) + x(i, i)) & \text{if } j < i \\ x(i+1, j) & \text{if } j = i \end{cases}$
3. **DAG**
   - Subproblems $x(i, j)$ only depend on strictly larger $i + j$ so acyclic
   - Solve in order of decreasing i, then decreasing j
   - Base case: $x(n+1, j) = 0$ for $j \in [1, n]$
4. **Evaluate**
   - Solution given by $x(1, 1)$
5. **Analysis**
   - (# Subproblems) $\times$ (work per problem) = $O(n^2) \times O(1) = O(n^2)$

Rubric:

- Same as previous problem.

SCRATCH PAPER 1. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to refer to the scratch paper next to the question itself.

SCRATCH PAPER 2. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to refer to the scratch paper next to the question itself.

SCRATCH PAPER 3. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to refer to the scratch paper next to the question itself.

SCRATCH PAPER 4. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to refer to the scratch paper next to the question itself.

SCRATCH PAPER 5. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to refer to the scratch paper next to the question itself.

SCRATCH PAPER 6. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to refer to the scratch paper next to the question itself.