1 Short answer - 18 points

Answer the following questions. You may **briefly** (no more than 2 sentences) justify your answers, but a complete proof is not required.

(a) For each of the following recurrences, do the following:

- Provide a **tight asymptotic upper bound**.
- **No partial credit. Draw a square around your final answer.**

(i)

$$A(n) = A(n/2) + A(n/3) + A(n/4) + n^2$$

Solution: Let's try the recursion tree analysis. The work at the root is n^2. The work at the next level is $\left(\frac{n}{2}\right)^2 + \left(\frac{n}{3}\right)^2 + \left(\frac{n}{4}\right)^2 = \frac{n}{4} + \frac{n}{9} + \frac{n}{16} = \frac{61}{144}n$. The point is, the total work at each level is decreasing and the total work in the tree is dominated by the n^2 term at the root.+ The tight asymptotic upper bound is $O(n^2)$. ■

(ii)

$$B(n) = 2B(n/4) + \sqrt{n}$$

Solution: This problem is exactly HW4P1b. Let's look atg this using the recursion tree method. The work at the first level is $\sqrt{n}$. The work at the second level is $2 \cdot \sqrt{\frac{n}{4}} = 2 \cdot \frac{1}{2}\sqrt{n} = \sqrt{n}$. Hence, the work stays constant at each level. The work of each level is $\sqrt{n}$, there are $\log_4 n$ levels. The tight asymptotic upper bound is $O(\sqrt{n} \log n)$. ■

(iii)

$$C(n, m) = C(n/2, m/3) + O(nm)$$

Solution: The work of ith level is $O(m/3^i n/2^i)$ and decreases, so this recurrence is dominated by root level which costs $O(mn)$, so the time complexity is $O(mn)$. ■

(b) Consider two numbers x and y, where B is the base, and x_1, x_0, y_1, y_0 are integers. How does Karatsuba's algorithm compute xy using three multiplications instead of 4?

Hint: Remember Karatsuba's algorithm broke n-digit integers x and y into two $m = n/2$ digit numbers: $x = x_1 B^m + x_0$ and $y = y_1 B^m + y_0$

Note: This is a short answer (no more than two sentences). Equations are allowed but everything needs to be concise.

Solution: Direct multiplication $xy = x_1 y_1 B^{2m} + (x_1 y_0 + x_0 y_1) B^m + x_0 y_0$ needs to compute four mutiplications $x_1 y_1$, $x_1 y_0$, $x_0 y_1$, $x_0 y_0$. Karatsuba only compute three multiplications $z_0 = x_0 y_0$, $z_1 = x_1 y_1$, $z_2 = (x_1 + x_0)(y_1 + y_0) - z_0 - z_1$ and recombine them to be $xy = z_1 B^{2m} + z_2 B^m + z_0$.

You need to show how the number of multiplications is reduced to get credit. Also, the actual equation was on the cheatsheet! But if you were unfamiliar with the algorithm looking trying to understand the equation during the exam would burn precious minutes which is why its important to study an algorithm even if it is included on the cheatsheet. If you had trouble with this problem it likely means you are looking at a concept and "understand" what you are looking at but aren't mastering the concept. That's something to consider when studyign for future exams. ■

2 Short answer II - 12 points

Answer the following questions. You *may* **briefly** (no more than 2 sentences) justify your answers, but a complete proof is not required.

(a) You are given two character sequences $A[0 \ldots n-1]$, $B[0 \ldots m-1]$. **What are the *minimum and maximum* alignment possible between these two sequences**? Assume mismatch cost (α) and insertion/deletion (δ) costs are both equal to 1.

Solution: There was a typo in the exam and we meant to have the exam test your ability to find the minimum and maximum alignment cost possible.

Given arbitrary m, n we have that the minimum alignment cost possible happens when the two strings are as aligned as possible, and thus only $|m-n|$ amount of insertions/deletions need to occur. On the flip side, the maximum alignment cost possible happens when the two strings are as mismatched as possible, and we incur a cost of n deletions into m insertions to get $m+n$ cost.

Alternate interpretations such as "maximum min alignment cost" in place of maximum alignment cost (giving expressions such as $\max(m, n)$) are given large amounts of partial credit. But in lectures it was emphasize that alginment costs are different than the edit distance or longest common subsequence algorithms. ■

(b) Recall in lecture/discussion we discussed the median of median (linear time selection) algorithm. The algorithm we discussed breaks a array into lists of size five. What if we break the array into lists of size 15. **What is the recurrence and asymptotic running time** for the this modified version of linear time selection

Solution: In the new MoMs algorithm with chunks of 15, at each step we first break the array into chunks of 15 and find the median for each chunk using brute force constant time. After taking the median of all the medians (by calling the algorithm recursively), we notice that half of the elements in 7/15 of the chunks are less/more than the pivot element, and half the median elements are less/more than the pivot element. Then at worst case, we eliminate $\frac{1}{2} \cdot \frac{n}{15} + \frac{1}{2} \cdot \frac{7n}{15} = \frac{8}{30}$ of the array, and we simply need to recurse on the other $\frac{22}{30}$ portion of the array. Finding this other portion requires a linear time partition algorithm. This gives us the following upper bound on recurrence

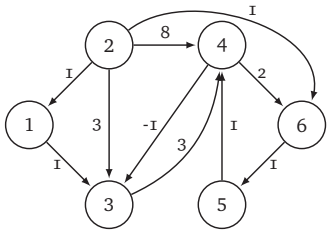
$$T(n) = T\left(\frac{n}{15}\right) + T\left(\frac{22}{30}\right) + O(n)$$

Using Master's Theorem or noticing that the fractions sum up to less than 1 gives us that the recurrence is dominated at root level, and we again arrive at an asymptotic runtime of $O(n)$. We went over this concept in exhaustive detail in the first 30 minutes of Lecture 11. Please refer to that if you are still struggling to understand the algorithm/recurrence. ■

3 Short answer III - 15 points

Answer the following questions. You *may* **briefly** (no more than 2 sentences) justify your answers, but a complete proof is not required.

(a) Consider the following graph:



We call the Floys-Warshall algorithm on this graph and fill out the three dimensional $d(i, j, k)$ matrix.
What is the value of $d(2, 4, 3)$?

Solution: In Floyd-Warshall algorithm, the vertices are labeled arbitrarily from 1 to n in any order. Then, $d(i, j, k)$ represents the weight of the shortest path from vertex i to vertex j in which the largest index of an intermediate vertex is k. The recursion for $d(i, j, k)$ can be written as

$$d(i,j,k) = min \begin{cases} d(i,j,k-1) \\ d(i,k,k-1) + d(k,j,k-1) \end{cases}$$

The graph in the question has no negative cycles. The vertices are labeled from 1 to 6. $d(2, 4, 3)$ is the weight of the shortest path from vertex 2 to 4 with potential intermediate nodes $\{1, 2, 3\}$. Hence, $d(2, 4, 3) = 5$. The path is $2 \to 1 \to 3 \to 4$. ■

(b) You have a directed graph $G = (V, E)$ with all positive edge weights. **Describe an algorithm that finds the shortest path between all pairs of vertices.** You can use any of the cheat sheet algorithms as a black box.

Solution: In this problem, we are supposed to find the distances of the shortest paths between all pairs of vertices. All the edges in G have positive weights. This means there are no negative cycles in the graph. So, we can use any of the three algorithms from the cheat sheet. They would have different time complexities. Let n and m be the number of vertices and edges in G, respectively.

Using Dijkstra's. Dijkstra's algorithm can find the shortest distance from a given vertex to all vertices in G. We can run this algorithm once for every vertex to find the shortest distances between all pair of vertices. With Fibonacci heap as priority queue, Dijkstra's runs in $O(m + n \log n)$. So, running it n times would have $O(mn + n^2 \log n)$ time complexity. For sparse graphs with $m = O(n)$, the time complexity would be $O(n^2 \log n)$. For dense graphs with $m = O(n^2)$, the time

complexity would be $O(n^3)$.

Using Bellman-Ford. We can also use Bellman-Ford to find the shortest distance from a vertex to all vertices in G and run it n times. The time complexity is $O(n^2m)$ which would become $O(n^3)$ for sparse graphs and $O(n^4)$ for dense graphs.

Using Floyd-Warshall. Floyd-Warshall finds the shortest distances between all pairs of vertices. The time complexity of this algorithm is $O(n^3)$.

Out of the three, using Dijkstra's is the best choice followed by Floyd-Warshall. ■

4 Dynamic Programming - 10 points

In class we discussed the longest increasing subsequence problem but just to recap: we are given a sequence of n integers and the goal is to find the longest increasing subsequence. We also know that we can find the length of the longest increasing subsequence in polynomial time. But how do we find the actual LIS (the actual values that make up the LIS).

Basically if the input is: $[6,3,5,2,7,8,1]$, the output should be: $[3,5,7,8]$ (The actual subsequence). **Provide an algorithm (or modify the existing LIS algorithm) that returns the longest increasing subsequence values.** I included the LIS code from lectures/labs below so you could save some time and only include the modifications to the original algorithm.

Solution: The newly added logic is highlighted in magenta. We introduce a $predecessor$ array to track the previous element in the LIS for each position. After computing the LIS length, the algorithm identifies the end of the LIS, then backtracks through the $predecessor$ array to collect the elements in the sequence. Finally, the list is reversed to return the LIS in the correct order.

```
LIS-ITERATIVE(A[1..n]):
  A[n+1] ← ∞
  Initialize array LIS[0..n−1, 0..n]
  Initialize array predecessor[0..n−1]
  for i ← 0 to n−1 do
      predecessor[i] ← −1 ⟨⟨Initialize predecessor array⟩⟩

  for j ← 0 to n do
      if A[0] ≤ A[j] then
          LIS[0][j] ← 1

  for i ← 1 to n−1 do
      for j ← i to n−1 do
          if A[i] ≥ A[j] then
              LIS[i, j] ← LIS[i−1, j]
          else
              if LIS[i−1, j] < 1 + LIS[i−1, i] then
                  LIS[i, j] ← 1 + LIS[i−1, i]
                  predecessor[j] ← i ⟨⟨Update predecessor⟩⟩
              else
                  LIS[i, j] ← LIS[i−1, j]

  end_pos ← arg max(LIS[n, :]) ⟨⟨Identify the endpoint of LIS⟩⟩
  Initialize lis_sequence ← []
  while end_pos ≠ −1 do
      lis_sequence.APPEND(A[end_pos])
      end_pos ← predecessor[end_pos]

  lis_sequence.REVERSE() ⟨⟨Reverse to get correct order⟩⟩
  return lis_sequence
```

■

5 Dynamic programming - 15 points

Assume you have a chain that is n links long that you need to sell. However, the value of the chain is not linearly proportional with the number of links. You are given an array $A[1 \ldots n]$ where $A[i]$ stores the price of a chain with n links (you can also assume $A[0] = 0$. no links = no chain = no value).

You look at the prices and realize that multiple smaller chains would be more valuable than the n-link chain you have right now. But you also know that dividing the chain requires you cut (and destroy) a link meaning that every division reduces the total number of links.

Show a dynamic programming algorithm that finds the maximum value you can obtain from a n-11link chain assuming you sub-divide it correctly.

Recurrence and short English description(in terms of the parameters):

Solution:

$$MCV(i) = \begin{cases} 0 & \text{if } i < 1 \\ \max(A[i], \max\limits_{1 \le j \le \lceil \frac{i}{2} \rceil} MCV[j-1] + MCV[i-j]) & otherwise \end{cases}$$

$MCV(i)$ is the maximum value that can be obtained from an i-link chain.

To get partial credit for recurrence, the recurrence must be somewhat well-defined. That is, the return value of a recurrence with some arbitrary parameters must be computable. Errors such as having unknown variables in the recurrence or the recurrence being incomplete may result in 0 credit for recurrence.

The English description is meant to help the reader understand what you are trying to compute with the recurrence, and therefore it must describe what the return value of the recurrence represents in terms of the parameters. Explaining how the recurrence works, or vaguely describing what each parameters are without stating what the recurrence is does not count as an English description. Before making regrade requests for English description, make sure you check the solutions for DP lab problems and understand what an English description should look like. ■

Memoization data structure and evaluation order:

Solution: We can define a 1-dimensional array $MCV[0..n]$ and fill out from $MCV[0]$ to $MCV[n]$.

To get credit for data structure, you should state the dimension of the data structure you would use, and this should be aligned with the recurrence. In other words, it must be obviously reasonable why you would use such data structure given the recurrence.

For evaluation order, you must clearly state in which order your data structure would be filled in, for each dimension of the data structure. Words like top-down, left-right do not count since the meaning of those can vary depending on how you align the

axis(as an exception, if the data structure is one-dimensional, left-right or right-left may be accepted). The evaluation order must be aligned with the dependency of the elements, so the evaluation order must be consistent with the recurrence. For instance, if $f(i,j)$ is defined based on $f(i-1,j)$ and $f(i,j+1)$, then f must be computed for the first parameter smaller than i and second parameter greater than j before it can be computed for (i,j), therefore it must be computed in increasing i and decreasing j order. ■

Return value:

Solution: $MCV[n]$

You must specify which element in the data structure contains the final answer. Taking the max over every entry of the data structure may be accepted as it does not increase the time complexity. You may get no credit for return value if the element with the final result cannot be determined from the recurrence and the data structure. ■

Time Complexity:

Solution: $O(n^2)$, since there are n subproblems and each subproblem takes $O(n)$.

You are graded based on the consistency with the recurrence and the data structure. If your recurrence runs in $O(n^3)$ then $O(n^3)$ counts as a correct runtime while $O(n^2)$ is incorrect. ■

Alternative solution(2-dimensional recurrence):

Solution:

Recurrence and short English description(in terms of the parameters):

$$MCV(i,j) = \begin{cases} 0 & \text{if } i < 1 \text{ or } j > i \\ \max(A[j] + MCV(i-j-1,j), MCV(i,j+1)) & otherwise \end{cases}$$

$MCV(i,j)$ represents the maximum value that can be obtained from an i-link chain when the length of each partition must be at least j.

Memoization data structure and evaluation order:
2-dimensional array $MCV[-1..n, 0..n+1]$. Evaluate in increasing i, decreasing j order.

Return value:
$MCV[n,0]$

Time Complexity:
$O(n^2)$ ■

6 Graphing Algorithm I - 15 points

In the traveling salesman problem, we are trying to find the path of smallest length that visits every vertex exactly once. For a normal graph this is a very difficult problem but for simple cases, an efficient solution is possible.

Suppose you had a directed acyclic graph (DAG) $G = (V, E)$ with all positive edge weights. Describe a efficient algorithm **that returns the value of** the shortest path that visits every vertex in the graph exactly once. Note that not every DAG has a path that visits every vertex once and so if there is no such path, your algorithm should return -1. The beginning and ending vertices can be any vertices in G.

Solution: For a DAG, there can be no path that visits all the vertices, or **exactly one** path that visits all the vertices. This is because for any DAG, if there is a path that covers all the vertices, there must be only one source and one sink, which is the starting and ending point for that path. There can't be more that one path that covers all the vertices, because as the source and sink are fixed, if there's more than one possible path, there will be a cycle. For example, say we have source -> A -> B -> C -> sink, and source -> C -> B -> A -> sink, there will be a loop between vertices A and C. With that being said, we just need to figure out if there exist the only one path, and calculate the length for that.
Algorithm:
Perform topological sorting on the graph, and check that for each adjacent vertices pair after the sorting, is there an edge connecting them. If there is, accumulate the edge weight, if there's any "disconnected" case, return -1. Runtime for this algorithm is linear time with respect to the number of vertices and edges per the runtime of topological sorting.
Wrong approaches:
(Worth no point) Running DFS / BFS give you no clue on a valid path as you are revisiting the vertices.
(Worth no point) Running shortest path algorithm give you no clue on a valid path as you don't have to visit all the vertices.
(Worth 5 points) Running topological sort and then run DFS / BFS or shortest path algorithms on the source node is still wrong, changes nothing from the above two points. ■

7 Graphing Algorithm II - 15 points

You are given a directed graph $G = (V, E)$ where every edge weight can be positive or negative and is marked as red or black. A red-black path is a path in the graph where edges alternate between red and black and can start with either a black or red edge.

Give an algorithm that gives you the shortest possible red-black path between vertices s and t.

Solution: Solution1

We build a graph $G' = (V', E')$ under the following rules:

(a) For each $v \in V$, we have $v_b, v_r \in V'$.

(b) For each $e \in E$ where $e = u -> v$, we have $e' = u_b -> v_r \in E'$ if e is a black edge, and $e' = u_r -> v_b \in E'$ if e is a red edge. The weight of e' = the weight of e.

With some attention, we notice that in this new graph G', any path of length greater or equal to 2 is a red-black path in G. Then, we utilize Bellman Ford method, as Dijkstra fails in graph with negative edges. We run Bellman Ford algorithm on G' twice starting from s_b and s_r. Since the shortest red-black path from s to t could starts/end at either black or red, we return the minimum value of $\{dist(s_b, t_b), dist(s_r, t_b), dist(s_b, t_r), dist(s_r, t_r)\}$. Constructing G' takes $O(V+E)$ time and running Bellman Ford twice takes $O(VE)$ time. So the overall runtime will be $O(VE)$. Note that one can avoid running Bellman Ford twice via a supernode, but this only reduce the work by a constant factor. ■

Solution: Solution2

We could also use DP to solve the problem. And the idea is similar to using Bellman-Ford, by adding an extra check before each iteration to compare the color between the previously used edge and the current edge. Note that simply mentioning modify Bellman Ford by having an extra if/else or so will not be awarded with full credit. As doing so will change the recurrence, thus change the DP table, filling order, and the return value. You need to explicitly state the recurrence of the new questions and analyze it following the steps for a DP question. If you applied this method, you will be graded using the standard rubric for Dynamic Programming Questions. We build a function Minpath(v,k,color) that will return the shortest red-black path from v to t using at most k edges and the start edge in this path must have color color. And then we have the following recurrence: Minpath(v,k,color) =

$$\begin{cases} 0 & \text{if } v = t \text{ and } k = 0, \\ \infty & \text{if } v \neq t \text{ and } k = 0, \\ \min \left\{ \begin{array}{ll} \min\limits_{\substack{vu \in E \\ \text{color}(vu) \text{ is } color}} & \{Minpath(u, k, color') + \ell(v, u)\}, \\ Minpath(u, k-1, color) & \end{array} \right\} & \text{else.} \end{cases}$$

Here, the variable $color$ will be either **black** or red. If $color$ = **black**, then $color'$ = red. If $color$ = red, then $color'$ = **black**.

We can memoize this function into a three-dimensional array with size $n \cdot m \cdot 2$. We can evaluate the array using three nested for-loops, one for each vertex v and one increasing j and another alternating between 0 and 1, aka between red and black. Then the total runtime is $O(nm)$, and we will return $min(Minpath(s, m, red), Minpath(s, m, black))$. In the above notes, n stands for the size of the vertex and m stands for the size of the edge.

One can optimize this data structure from 3-D to 2-D similar to Bellman-Ford method, but this does not improve any runtime efficiency, so we omit the tedious work.

Some common errors:

- One might try to color vertex by checking the color of in and out edges, which is incorrect, as a vertex can have both in/out black/red edges. So node can't be marked with any color.
- It's also not possible to build a graph that has ONLY red-black path. Consider the following graph: V = s,a,b,c,t; Eblack = (s,a),(s,c),(a,t),(b,t); Ered = (c,a),(a,b),(c,b),(t,c). Clearly there are three red-black paths from s-t: sabt,scat,scbt, any of them could be the shortest path. So we need to include all of these edges in the graph. But we notice that there will also be a s-t path "sat" that is not a red-black path. With this being said, building a graph with only red-black s-t path is not possible, as one might exclude some edges that are used by the true shortest red-black s-t path.
- Dijkstra's algorithm does not work in either solution as the graph contains negative edge weight. BFS does not work as the edges are not equally weighted. DFS does not work as it's not used in any shortest path questions.

■