# “CS/ECE 374 A”: Algorithms & Models of Computation, Spring 2025

# Midterm 2 Solution — April 14, 2025

| Name: | |
|---|---|
| NetID: | |

---

- Please *clearly PRINT* your name and your NetID in the boxes above.
- This is a closed-book but you are allowed a 1 page (2 sides) hand written cheat sheet that you have to submit along with your exam. If you brought anything except your writing implements, put it away for the duration of the exam. In particular, you may not use *any* electronic devices.
- **Please read the entire exam before writing anything.** Please ask for clarification if any question is unclear. The exam has 6 problems, each worth 10 points.
- **You have 150 minutes (2.5 hours) for the exam.**
- If you run out of space for an answer, continue on the back of the page, or on the blank pages at the end of this booklet, **but please tell us where to look.**
- **Write everything inside the box around each page.** Anything written outside the box may be cut off by the scanner.
- **Proofs are required only if we specifically ask for them.** You may state and use (without proof or justification) any results proved in class or in the problem sets unless we explicitly ask you for one.
- You can do hard things!
- **Do not cheat.** You know the student code and all that jazz. Grades do matter, but not as much as you may think, and your values are more important.

---

# 1 Sums and Recurrences

(a) Consider the recurrence

$$T(n) = n^{1/4} T\left(n^{1/4}\right) + n \quad n \geq 16, \qquad T(n) = 1 \quad 1 \leq n < 16$$

Give asymptotically tight bounds for the following, assuming the root of the tree is level one. No work is required; write your final answer in the box.

- (1 pt) Number of children at the second level:

  **Solution:** $n^{1/4}$ ■

- (1 pt) Work at the second level:

  **Solution:** $\sqrt{n}$ ■

- (1 pt) Depth of the recurrence:

  **Solution:** $\Theta(\log \log n)$ ■

- (1 pt) Value of the recurrence:

  **Solution:** $\Theta(n)$ ■

(b) Consider the recurrences below and give asymptotically tight bounds for each $T(n)$. No work is required; write your final answer in the box.

- (2 pts) $T(n) = 4T(n/2) + n$ for $n \geq 2$ and $T(1) = 1$

  **Solution:** $\Theta(n^2)$ ■

- (2 pts) $T(n) = 4T(n/2) + n^2$ for $n \geq 2$ and $T(1) = 1$

  **Solution:** $\Theta(n^2 \log n)$ ■

- (2 pts) $T(n) = 4T(n/2) + n^3$ for $n \geq 2$ and $T(1) = 1$

  **Solution:** $\Theta(n^3)$ ■

**Rubric:** The first two parts of (a) get 80% credit for answers consistent with treating the root as level 0 instead of level 1 ($n^{5/16}$ and $n^{3/8}$, respectively). All other parts are all-or-nothing.

## 2 Recursion/Divide and Conquer/Sorting/Selection

$A$ is an array of $n_1$ numbers that is sorted in *ascending* order. $B$ is an array of $n_2$ numbers sorted in *descending* order. You wished to create a sorted array by merging them but by mistake you concatenated $A$ with $B$ to create an array $C$ with $n = n_1 + n_2$ numbers (assume for simplicity that all the numbers are distinct). You do not have the original arrays anymore nor do you know $n_1$ and $n_2$. For example if $A = [1, 2, 6, 7]$ and $B = [8, 4, 3]$ then $C = [1, 2, 6, 7, 8, 4, 3]$. Describe an algorithm, as fast as possible, that given $C$, its size $n$, and a number $x$ checks whether $x$ is in $C$ or not.

**Solution:** We can obtain an $O(\log n)$-time algorithm via the following two step process. First, we find the "peak" of the array $C$, namely the largest number. More precisely we find its index $n'$ in $C$. We observe that $C[1..n']$ is a sorted array with numbers ascending and $C[n'..n]$ is a sorted array with numbers descending.

```
FindElement(C[1..n], x)):
  Let n' = FindIndexofMax(C[1..n])
  Use binary search on C[1..n'] to check if x is in C[1..n']
  if x found return YES
  else
      Use binary search on C[n', n] (with descending values) to check if x is in C[n'..n]
      return the outcome of the binary search
```

Now we describe a binary search like procedure to find the index of the maximum number in $C$.

```
FindIndexofMax(C[i..j]):
  Let n = j − i + 1
  If n < 10 use linear search (brute force) in O(1) time and return index
  mid = i + ⌊(j − i)/2⌋
  if (C[mid] < C[mid + 1]) return FindIndexofMax(C[mid + 1, j])
  else if (C[mid − 1] > C[mid] return FindIndexofMax(C[i, mid − 1])
  else return mid
```

The algorithm FindIndexofMax($C[1..n]$) takes $O(\log n)$ time since it is very similar to binary search. Thus FindElement($C[1..n], x$)) also takes $O(\log n)$ time since it is two binary searches at most after finding the index of the max. ■

**Rubric:** 10 points for a fully correct $O(\log n)$-time algorithm. Partial credit:

- 7 points for clearly articulating that we need to find the split point between $A$ and $B$, then searching each half, even if some of the finer details are missing.
- 4 points for attempting to use binary search in a reasonable way.
- Maximum 2 points for an $O(n)$ time algorithm.

## 3 Splitting Strings

Let $\Sigma$ be a finite alphabet and $L \subseteq \Sigma^*$ be a language. You have access to a routine IsStringinL($x$) that on input $x \in \Sigma^*$ returns whether $x \in L$ or not. Given $w \in \Sigma^*$ recall that $w \in L^*$ if and only if $w = w_1 w_2 \dots w_h$ for some $h \geq 1$ such that each $w_i \in L$; in this case we call $w_1 w_2 \dots w_h$ a valid $L$-split. In many applications we are interested in a $L$-valid split with some additional properties. Given $w$ and a split of $w$ into $w_1, w_2, \dots, w_h$ we define the cost($w_1, w_2, \dots, w_h$) to be $\sum_{i=1}^{h} |w_i|^2$.

Describe an algorithm that given $w \in \Sigma^*$ outputs the minimum cost of any $L$-valid split. Your algorithm should output $\infty$ if there is no $L$-valid split of $w$. You can assume that isStringinL($x$) takes $O(1)$ time in your analysis.

**Solution:** We solve this via DP.

To help the notation we define an auxiliary cost function on strings: for string $w$, $cost(w)$ is 1 if isStringinL($w$) $= 1$ (that is $w \in L$), otherwise $cost(w) = \infty$.

For $0 \leq i \leq n$ let $MinCostSplit(i)$ denote the cost of an $L$-valid split of $w[1..i]$ where $w[1..i]$ is the prefix of $w$ with the first $i$ characters. We write a recursive relation for $MinCostSplit$.

$$MinCostSplit(i) = \begin{cases} 0 & i = 0 \\ \min_{1 \leq j \leq i} MinCostSplit(i-j) + |j|^2 \cdot cost(w[(i-j+1)..i]) & i \geq 1 \end{cases}$$

The output is $MinCostSplit(n)$.

We can memoize the recursion using a one-dimensional array of size $n+1$ to store the values of $MinCostSplit(0), \dots, MinCostSplit(n)$ and we evaluate the array from $i = 0$ to $n$ in increasing order. To compute $MinCostSplit(i)$ requires $O(i)$ time based on the recursive formula and hence the total time to compute all the values in the array is $O(n^2)$. ■

**Standard dynamic programming rubric.** 10 points =

- 3 points for a clear and correct English description of the recursive function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.)
  - No credit if the description is inconsistent with the recurrence.
  - No credit if the description does not explicitly describe how the function value depends on the named input parameters.
  - No credit if the description refers to internal states of the eventual dynamic programming algorithm, like "the current index" or "the best score so far". The function must have a well-defined value that depends *only* on its input parameters (and constant global variables).
  - An English explanation of the *recurrence* or *algorithm* does not qualify. We want a description of *what* your function returns; points for an explanation of *how* that value is computed are assigned in other items.
- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error.
  - − 2 for greedy optimizations without proof, even if they are correct.

   - **No credit for iterative details if the recursive case(s) are incorrect.**

- 3 points for iterative details
  + 1 for describing an appropriate memoization data structure. **Hash tables are NOT an appropriate memoization data structure!**
  + 1 for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested for loops, be sure to specify the nesting order. (In particular, if you draw a rectangle for a 2d array, be sure to label and direct the row and column indices.)
  + 1 for correct time analysis. (It is not necessary to state a space bound.)

---

- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem specifically says otherwise.
- Iterative pseudocode is ***not*** required for full credit, provided the other details of your solution are clear and correct. We usually give two official solutions, one with pseudocode and one without.

  If your solution does includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. But you ***do*** still need and English description of the underlying recursive function (or equivalently, the contents of the memoization structure). ***Perfectly correct iterative pseudocode, with no explanation or time analysis, is worth at most 6 points out of 10***.
- Partial credit for incomplete solutions depends on the running time of the ***best possible*** completion (up to the target running time). For example, consider a solution that contains *only* a clear English description of a function, with no recurrence or iterative details. If the described function *can* be developed into an algorithm with the target running time, the solution is worth 3 points; however, if the function leads to an algorithm that is slower than the target time by a factor of $n$, the solution could be worth only 2 points (= 70% of 3, rounded).

# 4 Collecting Rewards

This is a variant of a problem from Homework 9.

Let $G = (V, E)$ be a directed graph in which each edge $e \in E$ has a non-negative reward $p(e)$ that can be collected by traversing it. Describe an algorithm that given $G$, a starting vertex $s \in V$ and an integer $k \geq 0$ computes the maximum reward that a walk starting at $s$ can collect if it is required to only collect reward from at most $k$ edges. Note the the reward on an edge $e$ is counted only the first time it is traversed in the walk since it is gone after it is picked up.

(a) (2 pts) How would you solve the problem if $G$ is strongly connected?

**Solution:** If $G$ is strongly connected then there is a walk in $G$ that starts in $s$ and visits all edges. Thus any set of $k$ edge rewards can be collected. Thus the optimum solution is simply output the sum of the largest $k$ edge rewards which can be found in $O(m)$ time by doing selection on the edge rewards. Sorting and computing the largest $k$ rewards takes $O(m \log m)$ time but that is also an acceptable solution. ■

**Rubric:** 2 points for identifying that we need to find the $k$ heaviest edges in the graph. No penalty for an algorithm that runs in $O(m \log m)$ time instead of $O(m)$.

(b) (8 pts) How would you solve the problem if $G$ is a DAG?

**Solution:** We use DP. We can first remove all vertices and edges that are not reachable from $s$ since they will not be part of any feasible solution. We can then do a topological sort of $G$ to get $v_1, v_2, \ldots, v_n$; note that we must have $v_1 = s$ since we removed all vertices not reachable from $s$. Note that in a DAG any walk starting at $s$ is in fact a path that starts at $s = v_1$ and visits the vertices in the walk in increasing order of the indices.

For $v \in V$ and $0 \leq h \leq k$ we let $MP(v, h)$ denote the max profit of a walk that picks at most $h$ edge rewards on a walk that starts in $s$ and ends in $v$. This satisfies the following recurrence:

$$MP(v,h) = \begin{cases} 0 & \text{if } v = s \text{ or } h = 0 \\ \max \left\{ \max \begin{pmatrix} MP(u,h), \\ MP(u,h-1) + p(u,v) \end{pmatrix} \middle| (u,v) \in E \right\} & \text{otherwise} \end{cases}$$

The output is $\max_{v \in V} MP(v, k)$.

The DP can be memoized using a 2-D array of size $n(k+1)$ where the entry in position $[i, h]$ stores $MP(v_i, h)$. The evaluation can be done by iterating over $h$ from 0 to $k$, and within each $h$ iterating $v$ in topological order (equivalently, iterating $i$ from 1 to $n$); note that this is *not* the only valid evaluation order. The time to compute each entry at a vertex $v$ is proportional to the in-degree of $v$. Thus for each fixed $h$ the time to compute $MP(v, h)$ for every $v$ is proportional to $\sum_v in-degree(v) = m$. Hence the total time for all $h$ is $O(mk)$. The initial topological sort takes $O(m + n)$ time, so the total time is dominated by $O(mk)$. ■

**Standard dynamic programming rubric.** 10 points =

- 3 points for a clear and correct English description of the recursive function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.)
  - No credit if the description is inconsistent with the recurrence.
  - No credit if the description does not explicitly describe how the function value depends on the named input parameters.
  - No credit if the description refers to internal states of the eventual dynamic programming algorithm, like "the current index" or "the best score so far". The function must have a well-defined value that depends *only* on its input parameters (and constant global variables).
  - An English explanation of the *recurrence* or *algorithm* does not qualify. We want a description of *what* your function returns; points for an explanation of *how* that value is computed are assigned in other items.
- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error.
  - − 2 for greedy optimizations without proof, even if they are correct.
  - **No credit for iterative details if the recursive case(s) are incorrect.**
- 3 points for iterative details
  - + 1 for describing an appropriate memoization data structure. **Hash tables are NOT an appropriate memoization data structure!**
  - + 1 for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested for loops, be sure to specify the nesting order. (In particular, if you draw a rectangle for a 2d array, be sure to label and direct the row and column indices.)
  - + 1 for correct time analysis. (It is not necessary to state a space bound.)

---

- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem specifically says otherwise.
- Iterative pseudocode is ***not*** required for full credit, provided the other details of your solution are clear and correct. We usually give two official solutions, one with pseudocode and one without.

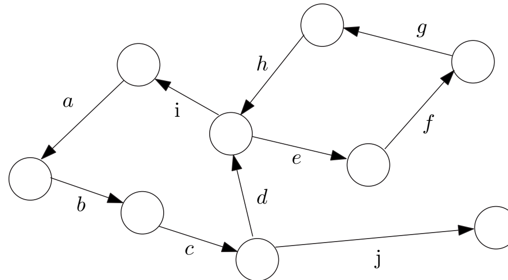
  If your solution does includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. But you ***do*** still need and English description of the underlying recursive function (or equivalently, the contents of the memoization structure). ***Perfectly correct iterative pseudocode, with no explanation or time analysis, is worth at most 6 points out of 10***.
- Partial credit for incomplete solutions depends on the running time of the ***best possible*** completion (up to the target running time). For example, consider a solution that contains *only* a clear English description of a function, with no recurrence or iterative details. If the described function *can* be developed into an algorithm with the target running time, the solution is worth 3 points; however, if the function leads to an algorithm that is slower than the target time by a factor of $n$, the solution could be worth only 2 points (= 70% of 3, rounded).

(Exercise for *after* the exam: combine these two parts to get an algorithm that works on an arbitrary directed graph.)

## 5 Graphs

Let $G = (V, E)$ be a directed graph and let $e_1 = (a, b)$ and $e_2 = (x, y)$ be two distinct edges. We wish to know if there exists a *path* that uses both $e_1$ and $e_2$, with $e_1$ appearing earlier in the path than $e_2$. Note that a path does not allow repetitions of vertices or edges while a walk allows both.

(a) Consider the following graph:




- (1 pt) Specify two edges $e_1$ and $e_2$ such that there is a path $P$ in the graph with $e_1$ before $e_2$ in $P$.

**Solution:** $e_1 = a$ and $e_2 = c$ ■

- (1 pt) Specify two edges $e_1$ and $e_2$ such that there is a *walk* $W$ in the graph with $e_1$ before $e_2$ in $W$ but there is *no* such *path* in the graph.

**Solution:** $e_1 = i$ and $e_2 = e$ ■

**Rubric:** One point for each. These are not the only correct answers.

(b) (4 pts) Describe an efficient algorithm that given $G = (V, E)$ and two distinct edges $e_1, e_2 \in E$ checks if there is a *walk* in $G$ with $e_1$ before $e_2$

**Solution:** Let $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$. There is a walk $W$ with $e_1$ before $e_2$ iff $u_2$ is reachable from $v_1$ in $G$. Thus the algorithm is simple.

- Use WFS from $v_1$ in the graph $G$.
- If $u_2$ is reachable from $v_1$ report YES, otherwise NO.

Running time is $O(n + m)$ time since we only use WFS on $G$. ■

**Rubric:** 4 points =

- 2 points for recognizing we have to check if $u_2$ is reachable from $v_1$.
- 1 point for applying WFS (or BFS/DFS).
- 1 point for run time analysis.

(c) (4 pts) Describe an efficient algorithm that given $G = (V, E)$ and two distinct edges $e_1, e_2 \in E$ checks if there is a *path* in $G$ with $e_1$ before $e_2$.

**Solution:** Let $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$. Since a path cannot reuse vertices, we can immediately say that there is no valid path if $u_1 = u_2$, $u_1 = v_2$, or $v_1 = v_2$. (It is fine to have $v_1 = u_2$, since then we can make a path consisting of just the edges $e_1$ and $e_2$.) As long as none of these cases happen, we just need a path from $v_1$ to $u_2$ that doesn't use $u_1$ nor $v_2$. This is now simply a reachability problem on the graph $G'$ we get by removing $u_1$ and $v_2$ (along with their adjacent edges) from $G$. Overall, this gives us the following algorithm:

- If $u_1 = u_2$, $u_1 = v_2$, or $v_1 = v_2$ return NO
- Obtain graph $G'$ from $G$ by removing $u_1$ and $v_2$, along with all incident edges
- Use WFS from $v_1$ in $G'$ to find all reachable nodes from $v_1$
- If $u_2$ is reachable from $v_1$ in $G'$ return YES, otherwise return NO

Computing $G'$ from $G$ and running WFS both take $O(n + m)$ time where $n = |V|$ and $m = |E|$. Thus total time is linear in the input size. ■

**Rubric:** 4 points =

- 1 point for checking if $u_1 = u_2$, $u_1 = v_2$, or $v_1 = v_2$. (Partial credit for only checking a subset of these.)
- 1 point for constructing $G'$ by removing $u_1$ and $v_2$. (Half credit for only removing one of these.)
- 1 point for applying WFS (or BFS/DFS).
- 1 point for run time analysis.

## 6 Shortest Paths with a Twist

Let $G = (V, E)$ be a directed graph where each edge $e$ has a non-negative length $\ell(e) > 0$. Each vertex $v \in V$ also has a non-negative value $a(v)$ specified in an array $A$. For simplicity, we will assume that the $a(v)$ values are distinct. You want to take a walk on this graph of total length at most $L$, starting and ending at a specified vertex $s$.

(a) (3 pts) Given $G, A, s, L$ and a specific vertex $t$, describe an efficient algorithm that checks if there is a walk of length at most $L$ that starts and finishes at $s$ and visits $t$.

**Solution:** We note that if we want to visit $t$, we should follow the shortest path to get from $s$ to $t$, and then the shortest path back. If sum of these two lengths is less than $L$, we can get to $t$; otherwise, we can't. This gives us the following simple algorithm:

- Use Dijkstra's algorithm in $G$ from source $s$ to compute $dist(s,t)$
- Use Dijkstra's algorithm in $G$ from source $s$ in $G^{rev}$ to compute $dist(t,s)$
- If $dist(s,t) + dist(t,s) \leq L$ output YES, otherwise NO

Total run time is $O(|E| \log |V|)$ for two invocations of Dijkstra's algorithm. ■

**Rubric:** 2 points for a correct algorithm, 1 point for runtime analysis. Note that for this part (but not the next), one can equivalently compute $dist(s,t)$ by running Dijkstra's starting from $t$ in $G$.

(b) (3 pts) Given $G, A, s, L$ describe an efficient algorithm to compute the highest-value of a vertex that any walk of length at most $L$ that starts and finishes at $s$ can visit.

**Solution:** The algorithm is essentially the same as the previous one.

- Use Dijkstra's algorithm in $G$ from source $s$ to compute $dist(s,v)$ for all $v \in V$
- Use Dijkstra's algorithm in $G$ from source $s$ in $G^{rev}$ to compute $dist(v,s)$ for all $v \in V$
- Let $Z = \{u \in V \mid dist(s,u) + dist(v,s) \leq L\}$
- Let $\alpha = \max\{a(u) \mid u \in Z\}$
- Output $\alpha$

Total run time is $O(|E| \log |V|)$ for two invocations of Dijkstra's algorithm. Computing $Z$ and $\alpha$ takes $O(n)$ time each. Thus the running time is dominated by the time for Dijkstra's algorithm. ■

**Rubric:** 2 points for a correct algorithm, 1 point for runtime analysis. Half credit for an algorithm that runs in time $O(|V| \cdot |E| \log |V|)$—eg, by running part (a) for every vertex.

(Continued on next page)

(c) (4 pts) Let $\alpha$ be the maximum value that you computed in the previous part. Describe an efficient algorithm to compute the second most valuable vertex your walk can reach subject to visiting the one with value $\alpha$. Your walk must still start and end at $s$ and have total length at most $L$.

**Solution:** Let $v^*$ be the vertex with $\alpha = a(v^*)$ such that $d(s, v^*) + d(v^*, s) \leq L$. To find the second highest valuable vertex we use graph reduction. The walk may visit this second highest vertex before $v^*$ or after. We consider the two case separately with the visit after $v^*$ being the first case.

- Construct graph $G_1 = (V \cup \{s'\}, E \cup \{(s', v^*)\})$ from $G$ by adding a new vertex $s'$ and adding new edge $(s', v^*)$ of length $d_G(s, v^*)$. The other edges in $G'$ have the same lengths as in $G$.
- Use Dijkstra's algorithm in $G_1$ from $s'$ to compute $d_{G_1}(s', v)$ for all $v \in V$.
- Let $\beta_1 = \max\{a(v) \mid v \neq v^*, d_{G_1}(s', v) + d_G(v, s) \leq L\}$
- Construct $G_2 = (V \cup \{s'\}, E^{rev} \cup \{(s', v^*)\}$ from $G$ by adding a new vertex $s'$ and adding new edge $(s', v^*)$ with length $d_G(v^*, s)$. Here $E^{rev}$ is the reverse of edges of $E$. The length of edges in $E^{rev}$ is the same as that in $E$.
- Use Dijkstra's algorithm in $G_2$ to compute $d_{G'}(s', v)$ for each $v \in V$
- Let $\beta_2 = \max\{a(v) \mid v \neq v^*, d_{G_2}(s', v) + d_G(s, v) \leq L\}$.
- Output $\max\{\beta_1, \beta_2\}$.

The running time of this algorithm is dominated by two Dijkstra computations on graphs that have only a constant number of additional edges and vertices compared to $G$ and hence the total run time is $O(|E| \log |V|)$. ■

**Rubric:** 3 points for a correct algorithm, 1 point for run time analysis.

This problem turned out to be more difficult that we expected, so *any* correct polynomial-time algorithm gets full credit. (For example, one can iterate over all possible vertices $v \neq v^*$ and check if there is a walk of length at most $L$ that vists $(s, v, v^*, s)$ or $(s, v^*, v, s)$ in that order. We can then take the maximum $a(v)$ for which this is true.