ECE 374 B: Algorithms and Models of Computation, Fall 2025

Midterm 2 – November 4th, 2025

- **You will have 75 minutes (1.25 hours) to solve all the problems. Most have multiple parts.** Don't spend too much time on questions you don't understand and focus on answering as much as you can! Make sure to check both sides of all the pages and make sure you answered everything. **Time is a factor!** Budget yours wisely.
- *No* resources are allowed for use during the exam except a multi-page cheatsheet and scratch paper on the back of the exam. ***Do not tear out the cheatsheet or the scratch paper!*** It messes with the auto-scanner.
- You should write your answers *completely* in the space given for the question. We will not grade parts of any answer written outside of the designated space.
- Please *use a dark-colored pen* unless you are *absolutely* sure your pencil writing is forceful enough to be legible when scanned. We will take off points if we have difficulty reading the uploaded document.
- Incorrect algorithms will receive a score of 0, but slower than necessary but correct algorithms will *always* receive some points, even brute force ones. Thus, *you should prioritize the correctness of your submitted algorithms over speed*; you will receive more points that way. On the other hand, submit the fastest algorithms that you know are correct; faster algorithms will receive more points.
- Any recursive backtracking algorithm or dynamic programming algorithm given without an *English* description of the recursive function (i.e., a description of the output of the function *in terms of their inputs*) will receive a score of 0.
- Any greedy algorithm or a modification of a standard graph algorithm given without a proof of correctness will receive a score of 0.
- For problems with a graph given as input, you may assume the graph is simple (i.e., it has no self-loops or parallel edges).
- Only algorithms referenced in the cheat sheet may be referred to as a "black box". You may not simply refer to a prior lab/homework for the solution and must give the full answer.
- Unless explicitly mentioned, **a runtime analysis is required for each given algorithm**.
- ***Don't cheat.*** If we catch you, you will get an F in the course.
- ***Good luck!***

Name: _____

NetID: _____

1 Short answer - 15 points

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

(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)

$$A(n) = nA(n-1) + n \qquad A(1) = 1$$

Solution: $n!$ **Provide visual explanation (with recursion tree) and explanation expanding out the recurrence ■

(ii)

$$B(n) = 2B(n/2) + n^2 \qquad B(1) = 1$$

Solution: This is directly from Quiz 07 (BYG). $O(n)$ ■

(b) For the recurrence $T(n) = aT(\frac{n}{4}) + \sqrt{n} \qquad T(1) = 1$, what is the minimum value of a for which the asymptotic bound of the recurrence would $= O(\sqrt{n}\log(n))$

Solution: So there are a couple interpretations for a which are reasonable and will be awarded full credit. The crux of the issue is that the question asks what is the minimum value of a for which the asymptotic bound is $O(\sqrt{n}\log(n))$. This would happen when $a = 2$ (see below).

But here's the thing, when $a = 1$ the asymptotic bound of the recurrence would be $= O(n)$. That's still bounded by $O(\sqrt{n}\log(n))$! The definition is of O-notation is that the function in the parentheses is a *upper*-bound. So if your recurrence is bounded by $O(n)$, then it is also bounded by $O(\sqrt{n}\log(n))$ and therefore $a = 1$ is still a legitimate answer. The same line of reasoning applies to $a = 0$. So $a = 0, 1, 2$ would all receive full credit. ■

Solution: Solution $a = 2$**:** Show that when a=2, the asymptotic bound is $O(\sqrt{n}\log(n))$. ■

Solution: Solution $a = 1$**:** Show that when a=2, the asymptotic bound is $O(\sqrt{n}\log(n))$. ■

Solution: Incorrect Solution $a \geq 2$**:** Show that when a=3, the asymptotic bound is ..... ■

(c) Imagine we have two sequences $\pi(x)$ which returns the x-th digit of π and $Fib(x)$ which returns the x-th digit of the Fibonacci sequence. Each of these functions take constant ($O(1)$) time to compute (it's a lookup table, no need to think about this too much).

Now we have the function below:

$$f(i,j) = \prod_{n=0}^{i} \pi(i + Fib(j)) + f(i-1,j) + f(i,j-1) \tag{1}$$

Assuming we can memoize this function perfectly, what is the asymtotic bound of the calculation for $f(n,n)$?

Solution: n^3 because there are n^2 problems and n work for each problem ■

2 Short answer (Recursion+DP) - 15 points

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

(a) A variation of this question was in Quiz 8.

I'd like to use the median-of-medians (MoM) algorithm but I don't want to write a function that finds the median value in a list of 5 values. Instead I break the input area into lists of **3** values, and choose the median of medians pivot that way. *Hint: the original MoM can be found in the cheatsheet*

(i) What is the recurrence that describes this new algorithm?

Solution: We first break a list of n elements into $\lfloor \frac{n}{3} \rfloor$ groups of size 3 each. Finding the median of each list through brute force takes $O(n)$ time. Finding the median of those $\approx n/3$ medians takes $T(\frac{n}{3})$ time. We then partition our array into a smaller and larger array, which takes $O(n)$ time, and have recursive calls for each partition. The runtime of the recursive calls is dominated by the larger partition, which is of size $T\left(\frac{3x-1}{4x}\right) = T\left(\frac{2}{3}\right)$, where $x = 3$ is the size of the lists we split into. We therefore arrive at our final recurrence for this algorithm:

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n)$$

■

(ii) What is the asymptotic running time of this version of MoM?

Solution: Drawing out the recursion tree shows that there is $O(n)$ work at each level. Therefore, total runtime is dependent on the number of levels in the tree, which is $O(\log n)$, since the largest subproblem shrinks at a rate of $\frac{2}{3}$. Therefore, we arrive at our final runtime:

$$T(n) = O(n \log n)$$

■

(b) Along similar lines, I'd like to use the QuickSort Algorithm but instead of randomly selecting the pivot (which is the usual implementation), I will use the median of medians method find a pivot. Let's use the original method for finding in the pivot (breaking the array in lists of size 5, then finding the median of each of those lists and then finding the median of those medians). Finding the pivot for an array of size n takes $O(n)$ time.

(i) What is the recurrence that describes this new algorithm?

Solution: This was discussed in Lecture 10. The exact recurrence appears in slide 31 but we discussed QuickSort recurrence before that and used it as a motivation to discuss the deterministic time selection algorithm.

Finding the pivot using MoM with a list size 5 takes $O(n)$ time. Partitioning the array based on the pivot takes $O(n)$ time. MoM with list size 5 guarantees that at least $\frac{3n}{10}$ of the elements are $\geq$ the pivot and at least $\frac{3n}{10}$ of the elements are $\leq$ than the pivot. Therefore, in the worst case, one partition has at least $\frac{3n}{10}$ elements and the other has at most $\frac{7n}{10}$ elements, and recurseivly sorting these takes $T\left(\frac{3}{10}\right) + T\left(\frac{7}{10}\right)$ time. We therefore arrive at our final recurrence:

$$T(n) = T\left(\frac{3n}{10}\right) + T\left(\frac{7n}{10}\right) + O(n) + O(n)$$

■

(ii) What is the **worst-case** asymptotic running time of this QuickSort algorithm?

Solution: Drawing out the recursion tree shows that there is $O(n)$ work at each level. Therefore, total runtime is dependent on the number of levels in the tree, which is $O(\log n)$, since the largest subproblem shrinks at a rate of $\frac{7}{10}$. Therefore, we arrive at our final runtime:

$$T(n) = O(n \log n)$$

This was also on your cheatsheet. ■

(c) Recall that the recurrence for the longest palindrome problem (you are given a sequence of number $A[1 \ldots n]$ and you need to find the longest palindromic sequence in A):

$$LPS(i,j) = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ \max \left\{ \begin{array}{c} LPS(i+1,j) \\ LPS(i,j-1) \end{array} \right\} & \text{if } i < j \text{ and } A[i] \neq A[j] \\ \max \left\{ \begin{array}{c} 2 + LPS(i+1,j-1) \\ LPS(i+1,j) \\ LPS(i,j-1) \end{array} \right\} & \text{otherwise} \end{cases}$$

In plain English, what does $LPS(i,j)$ represent?

Solution: This question format is directly from quiz 11 and the palindrome recurrence is in Lab11-P6 and Lab12P7.

$LPS(i,j)$ denotes the length of the longest palindrome subsequence of $A[i \ldots j]$. ■

3 Short answer (Graphs) - 20 points

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

(a) What is the maximum number of edges a directed-acyclic-graph with n vertices may have? I don't want a asymptotic bound. I want the actual value.

Solution: We discussed this problem at length in the lecture before the midterm. It's in the recording.

In any directed acyclic graph with n labeled vertices we can topologically sort the verticies such that

$$v_1, v_2, \ldots, v_n$$

and every edge is of the form $v_i \to v_j$ with $i < j$.

To avoid cycles, we can only include edges from earlier to later vertices in this ordering. For each pair $1 \leq i < j \leq n$, we may include the edge $v_i \to v_j$ at most once. The number of such pairs is

$$\binom{n}{2} = \frac{n(n-1)}{2}.$$

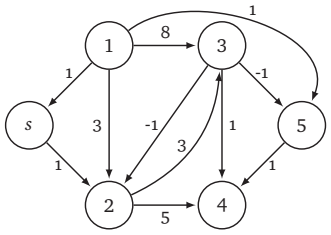
Taking the maximum number of edges yields the answer above. ■

Solution: Alternate explanation: The fact that the we have a DAG means that it can be sorted in topological order. Topological order means all the edges have to "flow" in one direction. Therefore, the first vertex i the topological order can be connected to $n-1$ vertices, the second vertex can be connected to $n-2$ vertices, and so on. Therefore we want the sum of all these edges:

$$\Sigma_i = 1^n(n-i) = \frac{n(n-1)}{2}$$

This is just Gausse's sum. Anything that is similar to $\frac{n^2}{2}$ recieved credit (that $\frac{1}{2}$ term is important though). ■

(b) Consider the following graph:



We call the Bellman-Ford algorithm on this graph (making node s the starting node) and fill out the two-dimensional $d(i,j)$ matrix.

What is the value of $d(1,3)$?

Solution: $d(1,3) = \infty$. Node 1 is unreachable from s , and distance values are intialized to infitinity, so $d(1,3)$ remains ∞, ■

(c) Given a directed graph G with n vertices and m unweighted edges, give an algorithm (as fast as possible - constants matter, not just asymptotes) that finds a vertex u, such that u is in the sink SCC of the meta-graph of G.

Solution: This is an alternative wording of a quiz question (Quiz 13, Problem type 1(e)).
As stated in the quiz manual:

- Calculate the reverse graph G^{rev}
- Run DFS with pre/post numbering on G^{rev}
- return the vertex with the largest post number

■

(d) Given a directed graph G with n vertices and m unweighted edges, we know node u is in the sink SCC of the meta-graph of G. Give an algorithm (as fast as possible - constants matter, not just asymptotic) that find the other vertices in the sink SCC of G.

Solution: This is also an alternative wording of a quiz question (Quiz 12, Problem type 1(d)).
In addition, we know that u is in the sink SCC.

- Run $S = Explore(G,u)$.
- Return S as the set of all vertices in the sink SCC containing u.

Additional note: since u is in the sink SCC, any path starting at u cannot leave this SCC, so we only need to run $Explore(G,u)$ once to visit the verticies in the SCC. ■

4 Dynamic programming - 15 points

Let's say you have at your disposable a wide assortment of (not necessarily dollar) coins and you need to make change for a particular value x. As you're doing so, you wonder to yourself: *what is the smallest number of coins you need to construct a total of x*. So let's formalize the question:

Problem: You are given a integer value x and an array A where each element of the array represents a coin denomination. Coins can be used multiple times.

Output: The smallest number of coins needed to make x. If there is no combination of coins that can make x, then the output should be 0.

Example: If $A = [1, 8, 9]$ and $x = 24$ output should be 3.

Here is some space so you can work out your solution before filling in the answer in the requested format on the next page:

Solution: This is a much simpler version of Lab 13 - Problem 1. At a high level, there can be two(not exhaustive) approaches.

- While keeping track of the remaining amount of change to make, recursively find the best coin to use.
- While keeping track of the remaining amount of change to make and the coin to consider, recursively find whether it is better to include the current coin or skip it and move to the next coin.

The first approach results in a 1-dimensional data structure, while the second results in a 2-dimensional data structure. ■

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

Solution:

- **1-D solution:** $C(i)$ represents the minimal number of coins required to make for i amount of change.

$$C(i) = \begin{cases} \infty & \text{if } i < 0 \\ 0 & \text{if } i = 0 \\ \min\limits_{1 \le k \le n} C(i - A[k]) + 1 & \text{otherwise} \end{cases}$$

- **2-D solution:** $C(i, j)$ represents the minimal number of coins required to make for j amount of change using the coins $A[1..i]$.

$$C(i,j) = \begin{cases} \infty & \text{if } j < 0 \text{ or } i \le 0 \\ 0 & \text{if } j = 0 \\ \min\big(C(i-1, j),\ C(i, j - A[i]) + 1\big) & \text{otherwise} \end{cases}$$

- To get full credit for English description, you must explain what $C(i)$ or $C(i, j)$ represents in terms of i and j, rather than explaining what i and j are or how the recurrence operates. The correctness of the description is judged based on whether the correct implementation of the provided English description would answer the given problem. You may get partial credit for writing a description that is reasonably close, depending on the grader's judgment.

- To earn partial credit above 4 for the recurrence, the recurrence you provide must be sufficiently relevant to the problem, subject to the grader's judgment. For example, your recurrence should include a parameter that reasonably keeps track of the remaining value to make. You may receive 1 point of partial credit for a recurrence that is irrelevant but still somewhat well-defined.

■

Memoization data structure and evaluation order:

Solution:

- **1-D solution:** Use a 1-dimensional array $A[1..x]$, evaluating entries in increasing order of the index.
- **2-D solution:** Use a 2-dimensional array $A[1..n, 1..x]$, evaluating entries in increasing order of both indices.
- You must provide either an understandable recurrence or a clear English description to receive credit, as the correctness of the data structure and evaluation order depends on the recurrence.
- To receive full credit, both the data structure and the evaluation order must be correct.
- Note that describing evaluation orders using phrases such as "left to right" or "top-down" without specifiying the orientation of the axis does not count, as these descriptions are ambiguous.
- If the dimensions of the data structure do not match the number of parameters

in the recurrence you provide, then you must explain how your recurrence maps onto the data structure in order to receive credit.

■

Return value:

Solution: $C(x)$ for 1-D approach, $C(n, x)$ for 2-D approach. You must provide either an understandable recurrence or a clear English description to receive credit, as the correctness of the return value depends on the recurrence. You must specify the exact cell of the data structure to be returned in order to receive credit. Using the variables employed in defining the recurrence such as $C(i, j)$ does not count, unless the meaning or values of these variables are explicitly specified.

■

Time Complexity:

Solution: Since we must iterate over n coins and the change amount less than or equal to x, the time complexity is $O(nx)$ for both approaches. You must provide either an understandable recurrence or a clear English description to receive credit, as the correctness of the time complexity depends on the recurrence. $O(n^2)$ is incorrect, as it does not capture the dependency on the amount of change to make.

■

5 Graphing Algorithms - 15 points

You are given a *undirected* graph $G = \{V, E\}$ with weighted (all positive) edge weights and two coins on vertices a and b. Every turn the two coins can only move across one edge. The weights on the edges represent tolls that the coins have to pay, but once each coin pays the toll once, they can use the edge again for free.

You want to find the vertex t that the two coins can meet. But there is a wrinkle, you need to minimize the sum of the tolls that both coins have to pay. Provide an algorithm that finds this minimal time.

Solution: This is a superficially modified version of Lab15-P3. and like in the lab, there are multiple solutions. Also like the lab, *the coins have to move every turn.* It was in the lab, we reinterated it in the classroom and it is in the problem description ■

Solution (Approach I: Parity Construction): Any sequence of k moves that bring the two coins to a common vertex x defines a walk of length $2k$ from a through x to b. Thus, we are looking for the shortest walk from a to b that uses even number of edges. We reduce to a standard shortest-path problem in a new graph $G' = (V', E')$ as follows:

- $V' = V \times \{0,1\} = \{(v, b) \mid b \in V \text{ and } b \in \{0,1\}\})$.
- $E' = \{(u, b)(v, 1-b) \mid uv \in E \text{ and } b \in \{0,1\}\}$. Edges in G' are undirected, because edges in the original graph G are undirected.

  For any walk $v_0 \to v_1 \to v_2 \to \cdots \to v_\ell$ in G, there is a corresponding walk $(v_0, 0) \to (v_1, 1) \to (v_2, 0) \to \cdots \to (v_\ell, \ell \bmod 2)$ in G'. Thus, every even-length walk from s to t in G corresponds to a walk from $(s, 0)$ to $(t, 0)$ in G' and vice versa.
- We set the edge weight of E' as the following: $\{l((u, b)(v, 1-b)) = l(u, v) | uv \in E \text{ and } b \in \{0,1\}\}$
- It maybe tempting to just explore the shortest-path distance in G' from vertex $(a, 0)$ to $(b, 0)$, but this might not always give the correct answer; as if coin A uses a certain edge once and wants to re-use it, the toll of that edge will be counted again in G'.
- To overcome this problem, what we can do is to assume every vertex might be the meeting point of the two coins. So for every vertex $k \times \{0,1\}$ in G', just calculate the shortest path from $(a, 0)$ to $k \times \{0,1\}$ in G', and $(b, 0)$ to $k \times \{0,1\}$ in G', both using Dijkstra. This gives us two shortest even length paths in G for each k, one denoting odd length from coin a or b to k, the other denoting even length from coin a or b to k. Then we just need to simulate all these $2|V|$ paths in G and find the one with actual shortest length in G. The simulation is needed as we do have a shortest path in G' that represents a shortest walk in G with certain parity, but the cost of that walk G' may not necessarily equivalent to the cost of the path in G.
- Constructing the graph by brute force takes $O(V' + E')$ time; running Dijkstra search twice takes $O(V' log V' + E')$; doing the simulation takes $O(V'E') = O(VE)$. Thus the resulting algorithm runs in $O(V' log V' + E' + V'E') = O(VE)$ time.

Notes:

(a) The graph has all positive non-uniform weights. So Dijkstra gives the best result. Using Bellman-Ford or Floyd-Warshall will receive some amount of partial credit; while using BFS/DFS will not.

(b) Approach that trying to modify any graph search algorithm without a proof will receive a 0.

(c) You can't run Dijkstra starting on a and b at the same time/simultaneously.

(d) Creating layers/masks representing paid and unpaid edges makes no sense. As during the graph traverse, there might be $4 \times 2^{|E|}$ different choices of certain edges being paid by neither A nor B/ A only/ B only/ both A and B. It is true that one can use this way as brute force to solve the problem, but a detailed solution is needed for credit. For example, an algorithm specifically explains a recursive function that can generate all possible walks that uses no more than $2|E|$ edges is considered an okay brute-force algorithm. As otherwise, any question could be answer with the phrase "Use brute force to solve".

(e) None of the graph search algorithm (BFS/DFS?Dijkstra/BF/FW) will guarantee a path with certain parity.

(f) It's not possible to have a 'dynamic' graph, with modified graph weight during the construction or traverse process. Modify the graph weight using graph search is equivalent to modifying an existed graph search algorithm, which worth 0 credit if no proof is demonstrated.

(g) Useful example graph, stored in an adjacent list:

$$
\begin{aligned}
&A : \{C : 1, K : 10\},\\
&C : \{A : 1, D : 1\},\\
&D : \{C : 1, E : 1\},\\
&E : \{D : 1, F : 1\},\\
&F : \{E : 1, B : 1\},\\
&B : \{F : 1, G : 10, K : 10\},\\
&G : \{J : 1, H : 1, B : 10\},\\
&H : \{G : 1, J : 1\},\\
&J : \{G : 1, H : 1\},\\
&K : \{A : 10, B : 10\}
\end{aligned}
$$

■

Solution (Approach II: Product Construction): We could also do product constructions. We reduce this problem to a shortest-path problem in an undirected graph $G' = (V', E')$ as follows:

- $V' = V \times V = \{(u, v) \mid u \in V \text{ and } v \in V\})$; the vertices of G' correspond to possible placements of the two coins.
- $E' = \{(u, v)(u', v') \mid uu' \in E \text{ and } vv' \in E\}$. The edges of G' correspond to legal moves by the two coins. Edges are undirected, because any move by the two coins

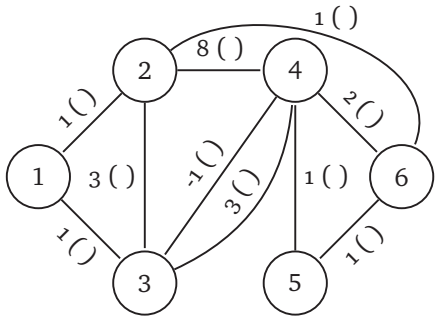
can be reversed.

- The weight of the edge $(u,v)(u',v')$ is set to the sum of the weights of edge uu' and vv', so $l((u,v)(u',v')) = l(uu') + l(vv')$
- We need to find the shortest-path distance from vertex (a,b) to any vertex of the form (v,v).
- First we compute the shortest-path distance from (a,b) to every vertex in G' that is reachable from (a,b) using Dijkstra's algorithm. Then a simple for-loop over the vertices of the input graph G finds the minimum distance to any marked vertex of the form (v,v). In particular, if no vertex (v,v) is reachable from (a,b), then no vertex (v,v) will be marked by Dijkstra, and so the algorithm will correctly report $\min \varnothing = \infty$.
- Once we have the shortest path from (a,b) to (v,v) for all $v \in V$ in G', we will do a similar simulation as the first approach to find the real shortest toll and meeting point in G.
- Constructing the graph by brute force takes $O(V'+E')$ time. Running Dijkstra once takes $O(V'logV'+E)$. The simulation takes $O(VE)$. Thus the resulting algorithm runs in $O(V'+E'+V'logV'+VE) = O(V^2+E^2+V^2logV^2+VE) = O(V^2logV+E^2)$ time.

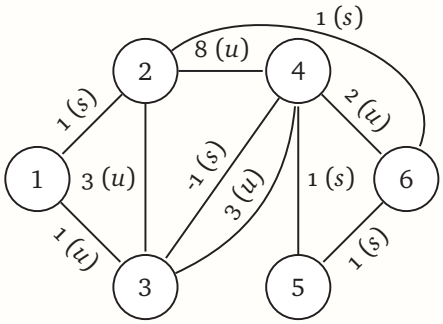
■

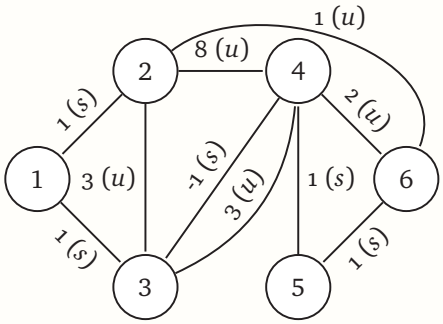
6 Minimum Spanning Trees - 10 points

For the following graph, label all the safe edges with a ("s"), all the unsafe edges with a ("u"), and all the edges which are neither with a ("n"):

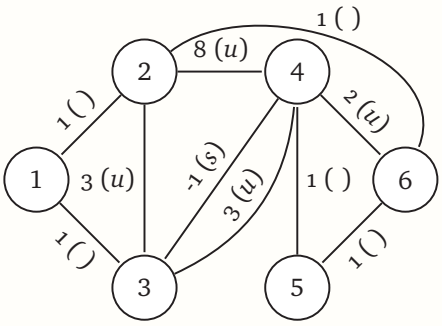


Solution: So as discussed in lecture, the key to this problem is to realize the the safe edges are all the edges that are part of the MST and the unsafe edges are the edges that are not part of the MST. But wait! As we discussed at the end of Lecture 18, if multiple edges have the same weight, you can simply add a bit of weight to each edge to prioritize some sides over the other. Knowing this, again, the edges that are part of the MST are the safe edges and the edges that are not part of the MST are the unsafe edges. Using this logic we have two options:

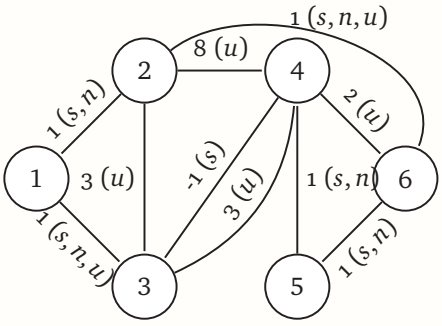




But let's say you forgot about the adding small weights to the edges to make them distinguishable. Let's use the definition of safe and unsafe given in lectures where a safe edge is one where the edge is the minimum edge cost in a particular cut and a unsafe edge is one where it is the maximum weight edge in a particular cycle. By this logic, all the non-1 edges are clearly defined as safe or unsafe:



Now the issue is what to label the 1-edges. It's true that since the edges aren't distinct, they don't fit neatly in the definition of safe edges because they are often tied for the minimum edge weight. There is one cycle containing the 1 edges and -1 edge that would make the 1-edges tie for the highest weight edge as well. So for the purposes of this midterm, you can label the 1-edges as safe or neither. Additionally, from the logic above, labeling {2,6} or {1,3} as unsafe is allowable as well.



Please refer to Gradescope for the actual rubric. We allowed to a lot of variation in the answers as long as the responses were semi logical. Illogical answers like marking {1,2} and {1,3} both as unsafe were marked wrong (how would 1 be connected to the spanning tree then?). ■

7 Recursion + Dynamic Programming + Graphs - 10 points

In class, we exhaustively discussed the Floyd-Warshall algorithm (shown below) and while we mainly focused on how to get the shortest path length, now we want to get the shortest path itself (the sequence of vertices). Below you are given the Floyd-Warshall algorithm that returns the minimum path length between any two vertices i and j. Notice that every time the minimum path length is updated, the intermediate node is recorded within the "Next" matrix.

Write an algorithm/function that reconstructs the minimum path (sequence of vertices) between i and j using this Next[] matrix.

```
//Initialize d array) for i = 1 to n do
   for j = 1 to n do
      d(i,j,0) = ℓ(i,j)
      (* ℓ(i,j) = ∞ if (i,j) not edge, 0 if i = j *)
      Next(i,j) = −1

//Compute length of shortest path
for k = 1 to n do
   for i = 1 to n do
      for j = 1 to n do
         if (d(i,j,k−1) > d(i,k,k−1) + d(k,j,k−1)) then
            d(i,j,k) = d(i,k,k−1) + d(k,j,k−1)
            Next(i,j) = k

//Detect negative cycles
for i = 1 to n do
   if (d(i,i,n) < 0) then
      Output that there is a negative length cycle in G
```

Solution: **Note:** We *only* have access to *Next*, s, and t. **No access** to d.

Path Recovery Intuition

Next(i,j) stores the vertex k through which the shortest $i \to j$ passes.

To recover shortest $a \to b$ path (*RSP*) #RSP(a, b)

- Find m s.t. $a...m...b$ is min $\#m = \text{Next}(a,b)$
- Find path from a to m #RSP(a, m)
- Find path from m to b #RSP(m, b)
- Concatenate

Core Algorithm: Incomplete, but a starting point

RSP(s, t) returns the shortest path from s to t

$$\text{RSP}(s,t) = \begin{cases} \text{RSP}(s,k) \cdot k \cdot \text{RSP}(k,t) & \text{where } k = \text{Next}(s,t) \end{cases}$$

Observation 1: If there is no $s \to t$ path, -1 appears in the output of RST(s,t)

$\nexists\, s \to t$ path

$\Rightarrow\ \forall k,\ d(s,t,k) = \infty$

$\Rightarrow\ \nexists\, k$ s.t. $d(i,j,k-1) > d(i,k,k-1) + d(k,j,k-1)$

$\Rightarrow\ \mathit{Next}(s,t) = -1$

$\Rightarrow\ -1 \in \text{RSP}(s,t)$

Close-ish Idea 1: Check $d(i,j,k-1) \neq \infty$. **Issue:** we don't have access to d.

Fix: Void output of RST if -1 occurs.

Observation 2: If an $s \to t$ path crosses a negative cycle, then $Next(i,i) \neq -1$ for some i.
$\exists$ negative cycle in G
$\Rightarrow$ $\exists i$ such that $\text{SP}(i,i) \ni v \neq i$
$\Rightarrow$ $\exists i$, $Next(i,i) \neq -1$
continued on next page... ■

Solution: continued
Failed Idea 1: Traverse diagonal of Next, abort on any non -1 entry. **Issue:** if the cycle is not on the s-t path, then our path is valid.
Close-ish Idea: Void RST output if repeated vertices are detected. **Issue: Hamiltonian Cycle?**
Fix Part 2: Run RST as normal. If Next(i,i) = -1 for any vertex in path. Void Output.

Observation 3: Worst case path is len-$|V|$, so RSP(s,t) should terminate in $|V|$ iterations.
Fix: Each call outputs 1 char. Add third variable h, bounded by $|V|$, to cap recursion depth.

Updated Recurrence
RSP(i, j, h) returns the shortest path from i to j with at most h intermediate vertices.
Requires: 1. unique i and j. **2.** $i \to j$ path exists, **3.** graph with no negative cycles

$$\text{RSP}(i,j,h) = \begin{cases} \varepsilon & \text{if } i = j \\ -1 & \text{if } h = 0 \text{ and } i \neq j \\ \text{RSP}(i,k,h-1) \cdot k \cdot \text{RSP}(k,j,h-1) & \text{where } k = \text{Next}(i,j) \end{cases}$$

Final Algorithm

```
function RecoverPath(s, t, |V|):
   if  s = t, output s
   P ← RSP(s, t, |V|−2)
   if  −1 ∈ P, output "No s → t path exists"
   if  ∃v ∈ P s.t.  Next(v, v) ≠ −1, output "Negative cycle in s → t path"
   output P
```

Runtime:
Line 1: RSP is depth-bounded by $|V|$. $O(|V|)$
Line 2: Traverse through path. $O(|V|)$
Line 3: $O(|V|)$ lookups in table
Total: $O(|V|)$

Note: 2AM Sumedh thought: If you post something without a stamp they return it to the sender's address. What if you put the recipients address under from and your address under to? Do you never have to pay for stamps again? ■

Problem 7 continued

EXTRA CREDIT (1 pt)

We know Richard E. Bellman is one of the authors of the Bellman-Ford algorithm but he came up earlier in lectures as the father of this computing paradigm. What is the computing paradigm he introduced?

Solution: Dynamic Programming. Also accepted memoization. ■

EXTRA CREDIT (1 pt)

Name a office hour time (Day of week+time) and the TA or CA that hosts that OH time.

Solution: Lots of answers. Look on ecealgo.com. ■

TA/CA name:

OH time:

This page is for additional scratch work!

This page is for additional scratch work!

ECE 374 B Algorithms: Cheatsheet

1 Recursion

Simple recursion

- **Reduction:** solve one problem using the solution to another.
- **Recursion:** a special case of reduction - reduce problem to a *smaller* instance of *itself* (self-reduction).

Definitions

- Problem instance of size n is reduced to *one or more* instances of size $n-1$ or less.
- For termination, problem instances of small size are solved by some other method as *base cases*

Arguably the most famous example of recursion. The goal is to move n disks one at a time from the first peg to the last peg.

Tower of Hanoi

Pseudocode: Tower of Hanoi

```
Hanoi (n, src, dest, tmp):
  if (n > 0) then
    Hanoi (n − 1, src, tmp, dest)
    Move disk n from src to dest
    Hanoi (n − 1, tmp, dest, src)
```

Divide and conquer

Divide and conquer is an algorithm paradigm involving the decomposition of a problem into the same subproblem, solving them separately and combining their results to get a solution for the original problem.

Sorting algorithms

Algorithm	Runtime	Space
Mergesort	$O(n \log n)$	$O(n \log n)$ $O(n)$ (if optimized)
Quicksort	$O(n^2)$ $O(n \log n)$ if using MoM	$O(n)$

We can divide and conquer multiplication like so:

$$bc = 10^n b_L c_L + 10^{n/2}(b_L c_R + b_R c_L) + b_R c_R.$$

We can rewrite the equation as:

Karatsuba's algorithm

$$\begin{aligned} bc = b(x)c(x) = (b_L x + b_R)(c_L x + c_R) = (b_L c_L)x^2 \\ + ((b_L + b_R)(c_L + c_R) - b_L c_L - b_R c_R)\, x \\ + b_R c_R, \end{aligned}$$

Its running time is $O(n^{\log_2 3}) = O(n^{1.585})$.

Recurrences

Suppose you have a recurrence of the form $T(n) = rT(n/c) + f(n)$.

The *master theorem* gives a good asymptotic estimate of the recurrence. If the work at each level is:

Decreasing:	$rf(n/c) = \kappa f(n)$ where $\kappa < 1$	$T(n) = O(f(n))$
Equal:	$rf(n/c) = f(n)$	$T(n) = O(f(n) \cdot \log_c n)$
Increasing:	$rf(n/c) = Kf(n)$ where $K > 1$	$T(n) = O(n^{\log_c r})$

Some useful identities:

- Sum of integers: $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
- Geometric series closed-form formula: $\sum_{k=0}^{n} ar^k = a\frac{1-r^{n+1}}{1-r}$
- Logarithmic identities: $\log(ab) = \log a + \log b, \log(a/b) = \log a - \log b, a^{\log_c b} = b^{\log_c a}$ $(a, b, c > 1)$, $\log_a b = \log_c b / \log_c a$.

Backtracking

Backtracking is the algorithm paradigm involving guessing the solution to a single step in some multi-step process and recursing backwards if it doesn't lead to a solution. For instance, consider the longest increasing subsequence (LIS) problem. You can either check all possible subsequences:

Pseudocode: LIS - Naive enumeration

```
algLISNaive(A[1..n]):
  maxmax = 0
  for each subsequence B of A do
    if B is increasing and |B| > max then
      max = |B|
  return max
```

On the other hand, we don't need to generate every subsequence; we only need to generate the subsequences that are increasing:

Pseudocode: LIS - Backtracking

```
LIS_smaller(A[1..n], x):
  if n = 0 then return 0
  max = LIS_smaller(A[1..n − 1], x)
  if A[n] < x then
    max = max {max, 1 + LIS_smaller(A[1..(n − 1)], A[n])}
  return max
```

Linear time selection

The *median of medians* (MoM) algorithms give a element that is larger than $\frac{3}{10}$'s and smaller than $\frac{7}{10}$'s of the array elements. This is used in the linear time selection algorithm to find element of rank k.

Pseudocode: Quickselect with median of medians

```
Median-of-medians (A, i):
  sublists = [A[j:j+5] for j ← 0, 5, . . . , len(A)]
  medians = [sorted (sublist)[len (sublist)/2]
             for sublist ∈ sublists]

  // Base case
  if len (A) ≤ 5 return sorted (a)[i]

  // Find median of medians
  if len (medians) ≤ 5
    pivot = sorted (medians)[len (medians)/2]
  else
    pivot = Median-of-medians (medians, len/2)

  // Partitioning step
  low = [j for j ∈ A  if j < pivot]
  high = [j for j ∈ A if j > pivot]

  k = len (low)
  if i < k
    return Median-of-medians (low, i)
  else if i > k
    return Median-of-medians (low, i-k-1)
  else
  return pivot
```

Dynamic programming

Dynamic programming (DP) is the algorithm paradigm involving the computation of a recursive backtracking algorithm iteratively to avoid the recomputation of any particular subproblem.

Longest increasing subsequence

The longest increasing subsequence problem asks for the length of a longest increasing subsequence in a unordered sequence, where the sequence is assumed to be given as an array. The recurrence can be written as:

$$LIS(i,j) = \begin{cases} 0 & \text{if } i = 0 \\ LIS(i-1,j) & \text{if } A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1,j) \\ 1 + LIS(i-1,i) \end{cases} & \text{else} \end{cases}$$

Pseudocode: LIS - DP

LIS-Iterative$(A[1..n])$:
  $A[n+1] = \infty$
  for $j \leftarrow 0$ to n
    if A[i] $\leq$ A[j] **then** $LIS[0][j] = 1$

  for $i \leftarrow 1$ to $n-1$ **do**
    for $j \leftarrow i$ to $n-1$ **do**
      if $A[i] \geq A[j]$
        $LIS[i,j] = LIS[i-1,j]$
      else
        $LIS[i,j] = \max\{LIS[i-1,j],$
                $1 + LIS[i-1,i]\}$
  return $LIS[n, n+1]$

Edit distance

The edit distance problem asks how many edits we need to make to a sequence for it to become another one. The recurrence is given as:

$$\text{Opt}(i,j) = \min \begin{cases} \alpha_{x_i y_j} + \text{Opt}(i-1, j-1), \\ \delta + \text{Opt}(i-1, j), \\ \delta + \text{Opt}(i, j-1) \end{cases}$$

Base cases: $\text{Opt}(i,0) = \delta \cdot i$ and $\text{Opt}(0,j) = \delta \cdot j$

Pseudocode: Edit distance - DP

$EDIST(A[1..m], B[1..n])$
  for $i \leftarrow 1$ to m **do** $M[i,0] = i\delta$
  for $j \leftarrow 1$ to n **do** $M[0,j] = j\delta$

  for $i = 1$ to m **do**
    for $j = 1$ to n **do**

$$M[i][j] = \min \begin{cases} COST\,[A[i]]\,[B[j]] \\ \qquad +M[i-1][j-1], \\ \delta + M[i-1][j], \\ \delta + M[i][j-1] \end{cases}$$

2 Graph algorithms

Graph basics

A graph is defined by a tuple $G = (V, E)$ and we typically define $n = |V|$ and $m = |E|$. We define (u, v) as the edge from u to v. Graphs can be represented as **adjacency lists**, or **adjacency matrices** though the former is more commonly used.

- *path*: sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $v_i v_{i+1} \in E$ for $1 \leq i \leq k-1$. The length of the path is $k-1$ (the number of edges in the path). *Note:* a single vertex u is a path of length 0.
- *cycle*: sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$ and $(v_k, v_1) \in E$. A single vertex is not a cycle according to this definition.
  Caveat: Sometimes people use the term cycle to also allow vertices to be repeated; we will use the term *tour*.
- A vertex u is *connected* to v if there is a path from u to v.
- The *connected component* of u, con(u), is the set of all vertices connected to u.
- A vertex u can *reach* v if there is a path from u to v. Alternatively v can be reached from u. Let **rch**(u) be the set of all vertices reachable from u.

Directed acyclic graphs

Directed acyclic graphs (dags) have an intrinsic ordering of the vertices that enables dynamic programming algorithms to be used on them.
A *topological ordering* of a dag $G = (V, E)$ is an ordering $\prec$ on V such that if $(u, v) \in E$ then $u \prec v$.

Pseudocode: Kahn's algorithm

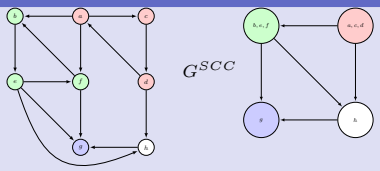
Kahn$(G(V,E), u)$:
toposort$\leftarrow$empty list
for $v \in V$:
  in$(v) \leftarrow |\{u \mid u \to v \in E\}|$
while $v \in V$ that has in$(v) = 0$:
  Add v to end of toposort
  Remove v from V
  for v in $u \to v \in E$:
    in$(v) \leftarrow$ in$(v) - 1$
return toposort

Running time: $O(n+m)$

- A dag may have multiple topological sorts.
- A topological sort can be computed by DFS, in particular by listing the vertices in decreasing post-visit order.

Strongly connected components

- Given G, u is *strongly connected* to v if $v \in$ rch(u) *and* $u \in$ rch(v).
- A *maximal* group of$G$: vertices that are all strongly connected to one another is called a strong component.



Pseudocode: Metagraph - linear time

Metagraph$(G(V,E))$:
  Compute rev(G) by brute force
  ordering $\leftarrow$ reverse postordering of V in rev(G)
    by **DFS**(rev(G), s) for any vertex s
  Mark all nodes as unvisited
  for each u in ordering **do**
    if u is not visited and $u \in V$ **then**
      $S_u \leftarrow$ nodes reachable by u by **DFS**(G, u)
      Output S_u as a strong connected component
      $G(V,E) \leftarrow G - S_u$

Running time: $O(m+n)$

DFS and BFS

Pseudocode: Explore (DFS/BFS)

```
Explore(G,u):
    for i ← 1 to n:
        Visited[i] ← False
    Add u to ToExplore and to S
    Visited[u] ← True
    Make tree T with root as u
    while ToExplore is non-empty do
        Remove node x from ToExplore
        for each edge (x, y) in Adj(x) do
            if Visited[y] = False
                Visited[y] ← True
                Add y to ToExplore, S, T (with x as parent)
```

- If B is a queue, *Explore* becomes BFS.
- If B is a stack, *Explore* becomes DFS.

Running time: $O(m+n)$

Pre/post numbering

Pre and post numbering aids in analyzing the graph structure. By looking at the numbering we can tell if a edge (u, v) is a:

- *Forward edge*: $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$
- *Backward edge*: $\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$
- *Cross edge*: $\text{pre}(u) < \text{post}(u) < \text{pre}(v) < \text{post}(v)$

Minimum Spanning Tress

- Tree = undirected graph in which any two vertices are connected by exactly one path.
- Sub-graph H of G is *spanning* for G, if G and H have same connected components.
- A minimum spanning tree is composed of all the safe edges in the graph
- An edge $e = (u, v)$ is a *safe* edge if there is some partition of V into S and $V \setminus S$ and e is the unique minimum cost edge crossing S (one end in S and the other in $V \setminus S$).
- An edge $e = (u, v)$ is an *unsafe* edge if there is some cycle C such that e is the unique maximum cost edge in C.

Pseudocode: Boruvka's algorithm: $O(m\log(n))$

```
T is ∅ (* T will store edges of a MST *)
while T is not spanning do
    X ← ∅
    for each connected component S of T do
        add to X the cheapest edge between S and V \ S
    Add edges in X to T
return the set T
```

Running time: $O(m\log(n))$

Pseudocode: Kruskal's algorithm: $(m+n)\log(m)$ (using Union-Find structure)

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty do
    pick e = (u, v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

Running time: $O((m+n)\log(m))$ if using union-find data structure

Pseudocode: Prim's algorithm: $(n)\log(n) + m$ (using Priority Queue)

```
T ← ∅, S ← ∅, s ← 1
∀v ∈ V(G) : d(v) ← ∞, p(v) ← ∅
d(s) ← 0
while S ≠ V do
    v = arg min_{u∈V\S} d(u)
    T = T ∪ {vp(v)}
    S = S ∪ {v}
    for each u in Adj(v) do
        d(u) ← min{ d(u), c(vu) }
        if d(u) = c(vu) then
            p(u) ← v
return T
```

Running time: $O(n\log(n) + m)$ if using Fibonacci heaps

Shortest paths

Dijkstra's algorithm:
Find minimum distance from vertex s to **all** other vertices in graphs *without* negative weight edges.

Pseudocode: Dijkstra

```
for v ∈ V do
    d(v) ← ∞
X ← ∅
d(s, s) ← 0
for i ← 1 to n do
    v ← arg min_{u∈V−X} d(u)
    X = X ∪ {v}
    for u in Adj(v) do
        d(u) ← min{(d(u), d(v) + ℓ(v, u))}
return d
```

Running time: $O(m+n\log n)$ (if using a Fibonacci heap as the priority queue)

Bellman-Ford algorithm:
Find minimum distance from vertex s to **all** other vertices in graphs *without* negative cycles. It is a DP algorithm with the following recurrence:

$$d(v,k) = \begin{cases} 0 & \text{if } v = s \text{ and } k = 0 \\ \infty & \text{if } v \neq s \text{ and } k = 0 \\ \min \begin{cases} \min_{uv \in E} \{d(u, k-1) + \ell(u, v)\} \\ d(v, k-1) \end{cases} & \text{else} \end{cases}$$

Base cases: $d(s, 0) = 0$ and $d(v, 0) = \infty$ for all $v \neq s$.

Pseudocode: Bellman-Ford

```
for each v ∈ V do
    d(v) ← ∞
d(s) ← 0

for k ← 1 to n − 1 do
    for each v ∈ V do
        for each edge (u, v) ∈ in(v) do
            d(v) ← min{d(v), d(u) + ℓ(u, v)}

return d
```

Running time: $O(nm)$

Floyd-Warshall algorithm:
Find minimum distance from *every* vertex to *every* vertex in a graph *without* negative cycles. It is a DP algorithm with the following recurrence:

$$d(i,j,k) = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } (i,j) \notin E \text{ and } k = 0 \\ \min \begin{cases} d(i,j,k-1) \\ d(i,k,k-1) + d(k,j,k-1) \end{cases} & \text{else} \end{cases}$$

Then $d(i, j, n-1)$ will give the shortest-path distance *from* i *to* j.

Pseudocode: Floyd-Warshall

```
Metagraph(G(V, E)):
    for i ∈ V do
        for j ∈ V do
            d(i, j, 0) ← ℓ(i, j)
                (* ℓ(i, j) ← ∞ if (i, j) ∉ E, 0 if i = j *)

    for k ← 0 to n − 1 do
        for i ∈ V do
            for j ∈ V do
                d(i, j, k) ← min{ d(i, j, k − 1),
                                  d(i, k, k − 1) + d(k, j, k − 1) }
    for v ∈ V do
        if d(i, i, n − 1) < 0 then
            return "∃ negative cycle in G"

    return d(·, ·, n − 1)
```

Running time: $\Theta(n^3)$