ECE 374 B: Algorithms and Models of Computation, Summer 2024

Midterm 2 – July 23, 2024

- **You will have 90 minutes (1.5 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.
- *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: _____

Date: _____

1 Short answer (2 questions) - 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) Give a *tight* asymptotic upper-bound for the following recurrences :

(i)

$$A(n) = 2A(n-1) + n/2 \qquad A(0) = A(1) = A(2) = A(3) = 1$$

Solution: Lab9 Page1 ■

(ii)

$$B(n) = 2B(n/2) + n \qquad B(0) = B(1) = 1$$

Solution: Lab9 Page2 ■

(b) Give the asymptotic running time that describes the following program assuming $n > 0$. (Hint: think about how the work grows with as the input increases, if you're thinking there's a typo, there isn't):

```
function foo(int n)
    if n > 100
        return 1
    else
        return foo(n+1) + foo(n+2) + foo(n+3)
```

Running time:

Solution: Despite three recursive calls, all paths reach the base case in a **constant** (≤ 100) number of steps. Any constant amount of work can be done in $O(1)$ time. ■

2 Short answer II (2 questions) - 10 points

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

(a) 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 9. What is the recurrence and running time for the this modified version of linear time selection

Solution:

With $\frac{n}{9}$ lists,

$$A_{\text{less}} : \text{mom} \leq \frac{1}{2} \cdot \frac{n}{9} = \frac{n}{18}$$

$$A_{\text{greater}} : \text{mom} \leq \frac{17n}{18}$$

$$T(n) \leq T\left(\frac{n}{9}\right) + \max\left\{T\left(|A_{\text{less}}|\right), T\left(\left|A_{\text{greater}}\right|\right)\right\} + O(n)$$

$$T(n) \leq T\left(\frac{n}{9}\right) + T\left(\frac{17n}{18}\right) + O(n)$$

$$T(n) = O(n)$$

■

(b) Give an example of a problem that has a recursive solution but that can not be efficiently memoized with dynamic programming.

Solution: Enumerating All Subsets of a Set: The output size is $2 \exp n$. The algorithm takes at least as much time as needed to create the output. No memoization will convert this from exponential time to polynomial time. ■

3 Recursion - 15 points

Given a decimal number as input, we need to write a program to convert the given decimal number into an equivalent binary number.

Input : 7
Output : 111

Input: 10
Output: 1010

Solution: `MakeTwosies(n)` converts a given positive decimal number to its binary representation. $\cdot$ means concat.

$$\texttt{MakeTwosies(n)} = \begin{cases} n & \text{if } n = 0, n = 1 \\ \texttt{MakeTwosies}(\lfloor \frac{n}{2} \rfloor) \cdot (n \mod 2) & \text{if } n > 1 \end{cases}$$

■

4 Dynamic programming - 15 points

A sequence is bitonic if it monotonically increases and then monotonically decreases. For example the sequences $\langle 1,4,6,8,3,2\rangle$ and $\langle 1,2,3,4\rangle$ are bitonic, but $\langle 1,3,12,4,2,10\rangle$ is not bitonic.

Given a sequence of n distinct integers ($\mathbb{A}$), give a efficient algorithm to find the largest bitonic subsequence in $\mathbb{A}$.

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

Solution: Define two recurrences as follows.
`MaxInc`(j) returns the length of the largest increasing sequence starting at index j.
`MaxDec`(j) returns the length of the largest decreasing sequence starting at index j.

$$\texttt{MaxInc(j)} = \begin{cases} 1+\texttt{MaxInc}(j-1) & \text{if } A[j-1] < A[j] \text{ and } j \geq 1 \\ 0 & \text{else} \end{cases}$$

$$\texttt{MaxDec}(j) = \begin{cases} 1+\texttt{MaxDec}(j+1) & \text{if } A[j] > A[j+1] \text{ and } j \leq n \\ 0 & \text{else} \end{cases}$$

■

Memoization data structure and evaluation order:

Solution: Define two 1D arrays: `MaxIncMemo[1...n]` and `MaxDecMemo[1...n]`.
For `MaxInc(j)`, `MaxInc(j-1)` must be pre-computed. Fill `MaxInc` from left to right.
For `MaxDec(j)`, `MaxDec(j+1)` must be pre-computed. Fill `MaxDecMemo` from right to left. ■

Return value:

Solution: $\max_{1\ldots n}\{\texttt{MaxIncMemo(j)} + \texttt{MaxDecMemo(j)}\}$. Let $\max(\emptyset) = 0$. ■

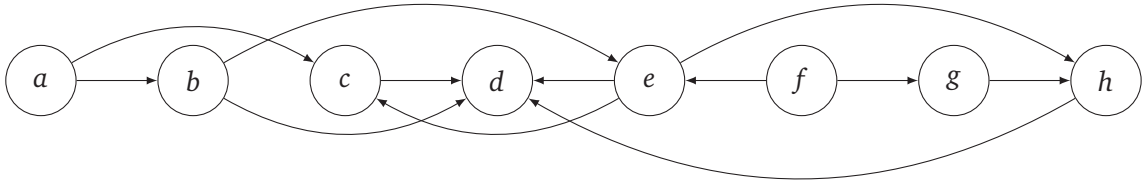
Time Complexity:

Solution: Analysis broken down into three $O(n)$ tasks.
`MaxInc(j)`: $O(n)$ calls, each $O(1)$. $O(n)$ to fill `MaxIncMemo`.
`MaxInc(j)`: $O(n)$ calls, each $O(1)$. $O(n)$ to fill `MaxDecMemo`.
Taking max over n elements: $O(n)$.
Total runtime: $O(n)$. ■

5 Short answer IV (4 questions) - 20 points

Answer the following questions. You *may* **briefly** (no more than 2 sentences) justify your answers, but a complete proof is not required. For the following graph problems, use the notation $G = (V, E)$, $n = |V|$ and $m = |E|$

(a) Consider the following graph:



What is the order of vertices visited using the vanilla Depth first search algorithm starting from vertex a. When there is a choice for which vertex to visit, choose the vertex according to alphabetical order:

Solution: I hate making tables in latex. Answer in bottom left most cell.

Path	Stack	To visit
		a b c d e f g h
a	c b	b c d e f g h
a b	c e d	c d e f g h
a b d	c e	c e f g h
a b d e	c h c	c f g h
a b d e c	c h d	f g h
a b d e c	c h	f g h
a b d e c h	c	f g
a b d e c h		f g
a b d e c h f	g	g
a b d e c h f g		

■

(b) You are given a **connected, undirected** graph and after running DFS you realize there are no backward, forward or cross edges! how many paths are there between any two vertices $u, v \in V$.

Solution: Cross edges, forward edges, backwards are non-tree edges. Whats left is a tree, which has one simple path between any two nodes. ■

(c) Give an example of a graph with atleast 5 nodes, that has the same BFS and DFS vertex traversals.

Solution: A linked list with 5 nodes. Use your preschool crayons to test it. ■

(d) What type of graph with n nodes has the few possible topological sorts?

Solution: Answer 1: A directed acyclic graph with n-nodes with a cycle has o topological orderings by definition. Smart Response.

Answer 2: Design a graph that minimizes the number of left to right orderings vertices without using left/backwards pointing edges. A linked list with n nodes has one way to do this. Enlightened response. ■

6 Short answer V (3 questions) - 15 points

Answer the following questions. You *may* **briefly** (no more than 2 sentences) justify your answers, but a complete proof is not required. For the following graph problems, use the notation $G = (V, E)$, $n = |V|$ and $m = |E|$

(a) I have a directed graph with all positive edge weights. Give an efficient algorithm to find the all shortest paths between any two vertices $i, j \in V$.

Solution: Run Dijkstra's $O(E + V \log V)$ algorithm from each vertex $O(V)$ times, using back pointers to store the predecessor for path recovery. Runtime: $O(VE + V^2 \log V)$.

Since $E \leq V^2$, this is $O(V^3)$. You might as well use Floyd-Warshall, with back pointers, which also runs in $O(V^3)$ and finds all pairs shortest paths. ■

(b) Given a directed graph $G = (V, E)$, how do we efficiently find all the vertices that can reach u.

Solution: Reverse all edges in G. Run DFS from u keeping track of tree edges. Return all vertices added to the spanning tree. ■

(c) You are given a directed graph $G = (V, E)$ but instead of the edges having weights, now the vertices have an associated cost and edges can no cost. Describe an efficient algorithm to find the path with the smallest cost (minimize the vertex costs you have to pay).

Solution: Assuming that costs are all positive, return the minimum cost vertex. Any additional vertex you visit will only increase the cost. $O(V)$ time to iterate through the vertex set. ■

7 Graph algorithm - 10 points

You are given three weighted, directed acyclic graphs ($F = (V, E_F)$, $G = (V, E_G)$ and $H = (V, E_H)$) that have the same vertices but different edges. Every edge is assigned an integer value that could be positive or negative.

You are given two nodes s and t. The length of a path in each graph is defined as $\ell_F(s,t)$ for F, $\ell_G(s,t)$ for G and $\ell_H(s,t)$ for H. You need to find the path that exists in both G and H with the minimum combined length (in other words, find minimum value of $\ell_F(s,t) + \ell_G(s,t) + \ell_H(s,t)$).

Solution: Given G_1, G_2, and G_3 are DAGs: Construct G' as follows:

$$V' = V$$
$$E' = E_1 \cap E_2 \cap E_3$$

Call DAG-SSSP from lecture starting at s.	$O(V+E)$
Return $d(s,t)$.	$O(1)$
Runtime: $O(V+E)$	■

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] = ∞
    for j ← 0 to n
        if A[i] ≤ 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]
                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:

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

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

Pseudocode: Edit distance - DP

```
EDIST(A[1..m], B[1..n])
    for i ← 1 to m do M[i, 0] = iδ
    for j ← 1 to n do M[0, j] = jδ

    for i = 1 to m do
        for j = 1 to n do
            M[i][j] = min { COST[A[i]][B[j]]
                            +M[i − 1][j − 1],
                          δ + M[i − 1][j],
                          δ + M[i][j − 1]
```

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, $\mathrm{con}(u)$, 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.

DFS and BFS

Pseudocode: Explore (DFS/BFS)

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

Note:

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

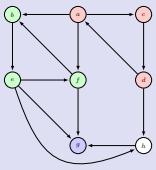
Pre/post numbering

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:

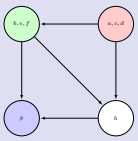
- *Forward edge*: pre(u) < pre(v) < post(v) < post(u)
- *Backward edge*: pre(v) < pre(u) < post(u) < post(v)
- *Cross edge*: pre(u) < post(u) < pre(v) < post(v)

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 nother is called a strong component.



G^{SCC}



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
      $G(V, E) \leftarrow G - S_u$

Shortest paths

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

Pseudocode: Dijkstra

for $v \in V$ **do**
  $d(v) \leftarrow \infty$
$X \leftarrow \varnothing$
$d(s, s) \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  $v \leftarrow \arg\min_{u \in V - X} d(u)$
  $X = X \cup \{v\}$
  for u in Adj(v) **do**
    $d(u) \leftarrow \min\{(d(u),\ d(v) + \ell(v, u))\}$
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 $v \in V$ **do**
  $d(v) \leftarrow \infty$
$d(s) \leftarrow 0$

for $k \leftarrow 1$ to $n - 1$ **do**
  for each $v \in V$ **do**
    for each edge $(u, v) \in$ in(v) **do**
      $d(v) \leftarrow \min\{d(v), d(u) + \ell(u, v)\}$

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 $i \in V$ **do**
    for $j \in V$ **do**
      $d(i, j, 0) \leftarrow \ell(i, j)$
        (* $\ell(i, j) \leftarrow \infty$ if $(i, j) \notin E$, 0 if $i = j$ *)

  for $k \leftarrow 0$ to $n - 1$ **do**
    for $i \in V$ **do**
      for $j \in V$ **do**
        $d(i, j, k) \leftarrow \min \begin{cases} d(i, j, k-1), \\ d(i, k, k-1) + d(k, j, k-1) \end{cases}$
  for $v \in V$ **do**
    if $d(i, i, n - 1) < 0$ **then**
      return "$\exists$ negative cycle in G"

  return $d(\cdot, \cdot, n - 1)$

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