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)

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

(ii)

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

(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 $= O(\sqrt{n}\log(n))$

(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 $f(n,n)$?

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) 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?

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

(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?

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

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

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

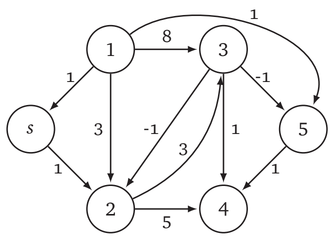
In plain English, what does $LPS(i,j)$ represent?

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.

(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 $d(1, 3)$?

(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.

(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.

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:

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

Memoization data structure and evaluation order:

Return value:

Time Complexity:

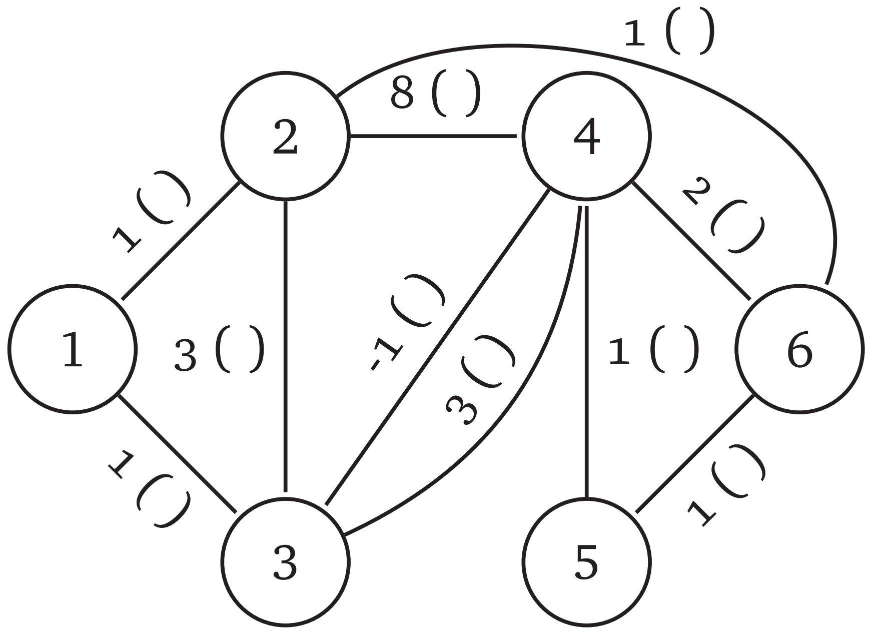
5 Graphing Algorithms - 15 points

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.

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"):



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
        $d(i, j, 0) = \ell(i, j)$
        (* $\ell(i, j) = \infty$ 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 $G$
```

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?

EXTRA CREDIT (1 pt)

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

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

Definitions

- **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).
  - 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*

Tower of Hanoi

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.

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

$$bc = b(x)c(x) = (b_L x + b_R)(c_L x + c_R) = (b_L c_L)x^2 + ((b_L + b_R)(c_L + c_R) - b_L c_L - b_R c_R)\,x + b_R c_R,$$

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 $\leftarrow 0, 5, \ldots,$ len($A$)]
    medians = [sorted (sublist)[len (sublist)/2]
                  for sublist $\in$ sublists]

    // Base case
    if len (A) $\leq$ 5 return sorted (a)[i]

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

    // Partitioning step
    low = [j for j $\in$ A  if j < pivot]
    high = [j for j $\in$ 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
            $M[i][j] = \min \begin{cases} COST[A[i]][B[j]] \\ \quad + M[i-1][j-1], \\ \delta + M[i-1][j], \\ \delta + M[i][j-1] \end{cases}$
```

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

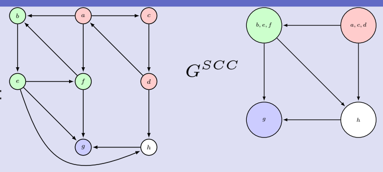
```
Kahn($G(V,E)$,$u$):
toposort$\leftarrow$empty list
for $v \in V$:
    in$(v) \leftarrow |\{u \mid u \to v \in E\}|$
while $v \in V$ that has in$(v) = 0$:
    Add v to end of toposort
    Remove $v$ from $V$
    for $v$ in $u \to v \in E$:
        in$(v) \leftarrow$ in$(v) - 1$
return toposort
```

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 $i \leftarrow 1$ 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 ToExplore is non-empty do
        Remove node $x$ from ToExplore
        for each edge $(x, y)$ in $Adj(x)$ do
            if Visited[$y$] $=$ False
                Visited[$y$] $\leftarrow$ 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*: 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)

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 $\varnothing$ (* $T$ will store edges of a MST *)
while $T$ is not spanning do
    $X \leftarrow \varnothing$
    for each connected component $S$ of $T$ do
        add to $X$ the cheapest edge between $S$ and $V \setminus S$
    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 $e = (u, v) \in E$ 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)

```
$T \leftarrow \varnothing, S \leftarrow \varnothing, s \leftarrow 1$
$\forall v \in V(G) : d(v) \leftarrow \infty, p(v) \leftarrow \varnothing$
$d(s) \leftarrow 0$
while $S \neq V$ do
    $v = \arg\min_{u \in V \setminus S} d(u)$
    $T = T \cup \{vp(v)\}$
    $S = S \cup \{v\}$
    for each $u$ in $Adj(v)$ do
        $d(u) \leftarrow \min \begin{cases} d(u) \\ c(vu) \end{cases}$
        if $d(u) = c(vu)$ then
            $p(u) \leftarrow v$
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 $v \in V$ 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: $\Theta(n^3)$