Dijkstra's Shortest Path Algorithm

PURPOSE: Find the Minimum Cost Path from a designated start node to a designated destination node in a graph *(directed or undirected)*

ASSUMPTIONS:

- graph is already stored, including:
  1) `n`, which is the number of graph nodes (where node are numbered 0 to N-1, not 1 to N)
  2) `edgeWeight` - stored as EITHER:
     - an adjacency matrix with 3 possible types of values:
       1) actual edge weights, 2) 0's in the diagonal, 3) "infinity" (i.e., MaxValue) for "no edge" node-pairs
       *[conventionally, **directed** graphs use rowNumber as the source ("from") & columnNumber as the sink ("to")]*
     - an array of adjacency lists
     -
- `GetWeight(a,b)` method returns the numeric `weight` value for edge `<a,b>` from the stored graph - for
  - internal AdjacencyMatrix: use direct address to get `edgeWeight[a,b]` (or `edgeWeight[a][b]`)
  - external AdjacencyMatrix: use direct address to read `weight` (after calculating offset, and `seek`-ing)
    [When calculating offset, allow for: a) headerRec; b) nodeNumbers being 0 to N=1, not 1 to N; c) weights being int or short or long or double… (as specified). So at start of program, calculate sizeOfHeaderRec, sizeOfARow, sizeOfAWeight once and for all to use in individual offset calculation each seek]
  - internal AdjacencyLists: search linkedList `[a]` for node `b` to get its weight *[not vice versa, so as to accommodate directed graph]*

- program has already gotten "from user": `startNode`**`Number`** & `destinationNode`**`Number`** (integers from 0 to N-1)
  *[if user instead provides startNode**Name** & destinationNode**Name**, then these have been converted into corresponding 2 **Numbers**]*

THE ALGORITHM

A) Initialize the 3 "working storage" arrays: *[This step must be done each time before step 2 is done.]*

- `included` – booleans *["Is **thisNode** **included** yet in the group of nodes already used to revise distance . . . or not?"]*
  - set all to false, except start's `included` is set to true

- `distance` - integers (usually) *["What's the **distance** from start to **thisNode** SO FAR?" (These are ceiling values that may go down).]*
  - set each to its corresponding `edgeWeight` from the graph for `start` to **thisNode**
    which would be either: 1) weight for an actual edge OR 2) 0 for `[start]` OR 3) "infinity" for no-edge cases

- `path` - integers *["What's the nodeNumber of **thisNode**'s predecessor on the **path** from start to **thisNode**?" (These are actual subscript values between 0 and N-1, corresponding to nodeNumbers).]*
  - set all to `-1's,` except use `startNodeNumber` instead when `distance[i]` has an actual edge weight (case #1 above)

B) Do the Search: *[Algorithm handles "normal case" – check if changes are needed for "special case", e.g., start == destination]*

```
while destination is NOT yet included {
1. out of nodes NOT yet included, choose target node (its node number) as
            the node with the minimum distance value                    [target is a subscript not a distance]
2. target now becomes included [as having been evaluated in step 3 below, to see it's effect]
3. check all distance values [they're ceilings] to see which ones can be lowered [i.e., loop: i = 0 to N-1]
          if included[i] is false                                       [GUARD #1 against doing the BIG TEST unnecessarily]
              if edgeWeight from target to i is a valid edgeWeight      [GUARD#2 against doing …]
                        [as opposed to a non-edge of 0 or "infinity"]
                                                                        [Finally comes the "BIG TEST" – i.e., should distance[i] ceiling be lowered?]
                    if distance[target] + edgeWeight from target to i   <   distance[i]
                    then: 1) distance[i] = distance[target] + edgeWeight from target to i
                          2) path[i]     = target
}
```

C) Report the answer:

- **TOTAL DISTANCE** of the minimum path from `start` to `destination` is found in `distance[destination]`
- **FINAL PATH** itself from `start` to `destination` is gotten from following the values in `path` array from `[destination]` to `-1`.
  However, this gives the path in reverse order, from `destination` to `start`.
  To instead correctly report the path from `start` to `destination`, either use:
  - recursion - printing the results on the way back UP recursion
  - OR push answers on a stack instead of printing them, then pop them off the stack to print them
  - OR store them to an array (incrementing, starting at 0), then print the array in reverse (decrementing, stopping at 0).