CSC2710 Analysis of Algorithms — Fall 2022
Unit 11 - Dynamic Programming

Book: §15.1 – 15.4

11.1 Dynamic Programming Overview

- Today we will look at a new algorithm design technique called dynamic programming[1].
- Dynamic programming is a technique for solving problems
  - It is very similar to divide and conquer in the sense that the a large problem is divided into smaller subproblems which must be solved.
  - In dynamic programming these subproblems generally overlap.
    - In other words, an overlapping subproblem is a subproblem that occurs more than once in the process of solving the original problem.
  - We will use extra space (a table) to allow us to solve each subproblem exactly once.
    - Specifically, once a problem is solved its solution is recorded in a table when the problem occurs again, the solution is just looked up in the table.
- Dynamic programming based solutions are generally applied to *optimization problems*.
  - An optimization problem is a problem where there are many possible solutions and we wish to find the optimal (for some definition of optimal) solution.
- For a dynamic programming solution to exist for a specific optimization problem, the problem must exhibit the *principle of optomality*.
  - *principle of optimality*: An optimal solution to any instance of an optimization problem is composed of optimal solutions to its subinstances.
  - This principal is sometimes also called the *optimal substructure property*.
- The development of a dynamic programming solution can be divided into four steps
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution in a bottom-up fashion.
  4. Construct an optimal solution from the computed information.
     - This step is omitted if we are only looking for the optimal value.

[1] Programming here has *nothing* to do with computer programming. It has everything to do with programming as a word for making decisions.

11.2 A Recursion Warm-up

- You have all, by this point in your career, seen the Fibonacci sequence.
  - e.g., $1, 1, 2, 3, 5, 8, \ldots$
- You may have seen this recursive definition before:

$$F(i) = \begin{cases} 1, & \text{if } i = 1 \text{ or } i = 2 \\ F(i-1) + F(i-2), & \text{if } i > 2. \end{cases}$$

- If I asked you to implement this you may just implement it as a top down recursive algorithm based on just the definition.
  - Here is such a CS 1 algorithm.

```
# Input: n is an integer
# Output: The nth Fibonacci number
def Fibonacci( n ):                                       # Line 1
    if( n == 1 or n == 2 ):                               # Line 2
        return 1                                          # Line 3
    return Fibonacci( n - 1 ) + Fibonacci( n - 2 )        # Line 4
```

- What's the recurrence relation for the running time of this algorithm?
  - We have the cost of 1 addition per operation and the cost of solving two subproblems one of size $n-1$ and one of size $n-2$.
  - This gives us the recurrence:

$$T(n) = \begin{cases} \Theta(1), \text{if } n \in \{1, 2\} \\ T(n-1) + T(n-2) + \Theta(1), \text{otherwise.} \end{cases}$$

  - Observe that we can lower bound our recursion by, $T(n) \geq 2T(n-2) + \Theta(1)$ for all $n > 2$ – this makes our recurrence easier to solve.
    * Simply use back-substitution.

$$\begin{aligned} T(n) &\geq 2T(n-2) + 1 \\ &= 2(2T(n-4) + 1) + 1 \\ &= 2(2(2T(n-6) + 1) + 1) + 1 \\ &\vdots \\ &= 2^k T(n-2k) + \sum_{i=0}^{k-1} 2^i \\ &= 2^{\frac{n-1}{2}} + \sum_{i=0}^{n-2} 2^i && \text{(Solving } n - 2k = 1 \text{ for } k.\text{)} \\ &= 2^{\frac{n-1}{2}} + 2^{n-1} - 1 && \text{(Closed form of geometric series.)} \\ &\in O(2^n). \end{aligned}$$

    * Yuck!
    * We can also work towards finding an upper bound by observing $T(n) \leq 2T(n-1) + 1$.
      · If we solve this recurrence we end up with $O(2^n)$ as a crude upper bound.

      - The actual upper bound is: $\Theta(e^n)$, where e is the natural number [2]. Notice this is the tight bound and is related to generator functions and the Golden ratio.

- Let's speed up the algorithm by making a space-time tradeoff
  - Observe that our algorithm recomputes a lot of the same values.
  - How about we remember previously computed solutions.
    - This is called memoization.
    - Memoization uses a table to store previously computed values. When a call is made first check the table if the answer is in the table use it otherwise compute it and store it in the table.
- Our memoization[3] trick will only work if our algorithm is top-down (i.e. works from n towards the base case.)
  - Here is such a solution with the assumption there is an array `memo` which is initialized to all `None`.

```
global memo
memo = [ None for x in range( 100 ) ]

# Input: n >= 1
# Output: The nth Fibonacci number
def FibonacciMomoized( n ):
    global memo
    if memo[ n - 1 ] != None:                                                # Line 1
        return memo[ n - 1 ]                                                 # Line 2
    if( n <= 2 ):                                                            # Line 3
        v = 1                                                                # Line 4
    else:                                                                    # line 5
        v = FibonacciMomoized( n - 1 ) + FibonacciMomoized( n - 2 )          # Line 6
    memo[ n - 1 ] = v                                                        # Line 7
    return v                                                                 # Line 8
```

- What's the running time?
  - Observe that we only recurse on the first time we compute $\textsc{Fib}(n)$ for all n.
    - Ignoring the cost of recursion how many subproblems must we solve?
      - Answer: n.
    - Ignoring the cost of recursion, how long does it take to solve a subproblem?
      - Answer: $\Theta(1)$ we only have to pay the cost of the addition and branching logic.
  - Any repeated call will look up the answer in the table which is instaneous ($\Theta(1)$ time).
  - The running time for our algorithm can be computed by observing it is simply the number of sub-problems times the cost to solve a subproblem which in this case $n \times \Theta(1) = \Theta(n)$.
- As it turns out, any recursive algorithm can be agumented to use memoization.

[2] Note that this implies the our hidden constant is proportional to $\frac{e}{2}$ in our crude upper bound.
[3] That word is *not* spelled wrong.

11.3 Developing Dynamic Programming Problems

- So far I have told you that Dynamic Programming is all about solving optimization problems.
  - Obviously the Fibonacci numbers problem is not a Dynamic Program. However it gives us the a key intuition about constructing dynamic programs.
  - In particular we get another intuition about Dynamic Programming: *Dynamic programming involves recursion with memoization.*
- The book develops a dynamic programming solution using four steps
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution in a bottom-up fashion.
  4. Construct an optimal solution from the computed information.
     - This step is omitted if we are only looking for the optimal value.
- I recently came aware of a breakdown I like better due to Demaine. Demaine asserts that we should break our dynamic programming constructions into five steps.
  1. Define the subproblems
     - What are the number of subproblems?
     - How do you denote a specific subproblem and what is it?
  2. Guess at part of the solution.
     - What are the number of choices for a solution to a particular subproblem?
     - Basically we guess at all possible solutions and select the *best* one.
  3. Relate the subproblem solutions to each other (i.e., develop recurrence)
     - Compute the time per subproblem
  4. Use a method to speed up the recurrence
     - Memoize or bottom-up table method.
       * By *memoize* we mean remember the previously computed answers in some form of table.
     - Compute the time per subproblem times the number of subproblems.
  5. Solve the original problem (just a specific subproblem really)
     - May need to combine subproblem solutions this will result in more time.

11.3.1 A First Problem – Coin Collecting Robot

- The first problem we will see is called the "Coin-Collecting Problem"[4]
  - This is a favorite problem of mine drawn from Anany Levitin's Analysis of Algorithms book.

  Definition 1 (Coin-Collecting Problem)**.**
  Input: *An $n \times m$ board with no more than one coin at every cell and a coin-collecting robot in the upper left-hand corner of the board.*
  Output: *Determine the maximum number of coins the robot can collect and the path required to collect the coins. The movement of the the robot is subject to three rules*

  1. *The robot must stop in the bottom right corner of the board.*
  2. *The robot can move at most one cell right or one cell down in a single step.*
     - *Visiting a cell that contains a coin results in the robot collecting the coin.*

- Let's apply our steps to arrive at a Dynamic Program to solve this problem.
- **Define Subproblems**: Denote by $F(i,j)$ the largest number of coins the robot can collect and bring to cell (i,j)
  - We should note that this gives rise to nm subproblems.
- **Guess at Solutions**:
  - Our guess is nothing more than what previous cell we used to reach our current location.
  - The cell (i,j) can be reached in one of two possible ways:
    1. Via cell $(i-1,j)$ (the cell above)
    2. Via cell $(i,j-1)$ (the cell to the left)
  - What are the largest number of coins brought to each cell?
    * $F(i-1,j)$ and $F(i,j-1)$ respectively.
    * Does everyone see why?
- **Relate Subproblems**: It stands to reason that the largest number of coins that the robot can bring to cell (i,j) is given by the maximum of $F(i-1,j)$ and $F(i,j-1)$ plus a possible additional one at cell (i,j).
  - The recurrence relation that defines a solution to this problem is given by:

$$F(i,j) = \begin{cases} \max\{F(i-1,j), F(i,j-1)\} + c_{ij}, & \text{If } 1 \le i \le n \text{ and } 1 \le j \le m \\ 0, & \text{If } i = 0 \text{ and } 1 \le j \le m \text{ or} \\ & j = 0 \text{ and } 1 \le i \le n \end{cases}$$

    Where,
    * c_{ij} is one if there is a coin in cell (i,j) and zero otherwise.
    * The robot is not allowed to overstep a boundary.
  - We note that at every step there is at most two subproblems to solve.
- **Speed-up Recursive Algorithm**: To solve this problem we will formulate a top-down solution using memoization

[4] These problems come from "Introduction to the Design and Analysis of Algorithms" by Anany Levitin

  - **Practice**: What does it mean when we solve the problem top-down?

- We will make use of an $m \times n$ table which we can fill either
  - row-wise
  - column-wise

- Let's look at the row-wise solution using a formal algorithm with memo $\mathit{memo}\,[1..n, 1..m]$

```
# Coin row problem
memo = [ [ 0 for x in range(1000) ] for y in range(1000) ]

# Input: i and j are indices into C, a 2D array of integers
# Output:
def CC( i, j, C ):
    global memo
    n = len(C)                                                  # Line 1
    m = len( C[i] )                                             # Line 2
    if( memo[i][j]!= None ):                                    # Line 3
        v = memo[i][j]                                          # Line 4
    elif( 1 <= i and i <= n and 1 <= j and j <= m ):            # Line 5
        v = max( CC(i-1, j, C), CC(i, j-1, C) + C[i][j] )       # Line 6
    elif( i == 0 and i <= j and j <= m ):                       # Line 7
        v = 0                                                   # Line 8
    elif( j == 0 and 1 <= i and i <= n ):                       # Line 9
        v = 0                                                   # Line 10
    memo[i][j] = v                                              # Line 11
    return v                                                    # Line 12
```

- **Solution**: To solve the problem we will make the call $\textsc{CoinCollect}\,(n, m)$.

- What is the time efficiency?
  - How many subproblems must we solve in the worst case?
    * In the worst case we must fill out every cell in the memo table which is of size $\Theta\,(mn)$
  - How long does it take to solve a subproblem (ignoring recursion)?
    * Ignoring the cost of recursion, it only takes $\Theta\,(1)$ time to solve a subproblem (cost to perform addtion and deal with the branching).

- If we want the path we must work backward from cell (n, m).
  - The whole idea rests on one observation, there is only 2 possible ways to arrive at cell (i, j) for any i and j with-in the grid boundaries.
  - If $\mathit{memo}\,[i-1, j] > \mathit{memo}\,[i, j-1]$ then the path to (i, j) came from above.
  - If $\mathit{memo}\,[i-1, j] < \mathit{memo}\,[i, j-1]$ then the path to (i, j) came from the left.
  - If $\mathit{memo}\,[i-1, j] = \mathit{memo}\,[i, j-1]$ then either direction is optimal (and thus valid).

- If we ignore ties, we can recover the path in $\Theta\,(n + m)$ time.
  - **Practice**: Why?

- **Practice**: Do you see the overlapping subproblems in both of the example problems?

11.3.2 Rod Cutting

- Let's consider another problem that can be solved efficiently by using Dynamic Programming.
  - The problem is called *Rod Cutting*.
  - It can be found in CLRS third edition.
- We can define the problem as follows:

  Definition 2 (Rod-Cutting Problem)**.**
  Input*: Given a rod of length n inches and a table of prices $p[1..n]$ where $p[i]$ gives the value of a rod of length i inches.*
  Output*: The maximum revenue r_n obtainable by cutting up the rod and selling the pieces.*

- Let's apply our steps to arrive at a Dynamic Program to solve this problem.
- **Define Subproblems**: Denote by $r(n)$ the maximum revenue that can be obtained by cutting a rod of length n. The subproblems are all possible single cuts you can make to that
- **Guess at Solutions**: We guess where to make a single cut to split the section of rod into two pieces.
- **Relate Subproblems**: The maximum revenue for $r(n)$ that can be obtained by cutting the rod once into a piece of length i and length $n-i$ such that we have the maximum revenue $r(n-i)$.
  - Notice we will look at all possible single cuts we can make for a rod of length n.
  - The recurrence for $r(n)$ is:

$$r(n) = \begin{cases} \max_{1 \le i \le n} \{p[i] + r(n-i)\} & \text{If } n > 0 \\ 0 & \text{Otherwise.} \end{cases}$$

    * In words, the maximum revenue is given by the best location to cut the rod in two.
    * This results in a rod of length i which we sell and of length $n-i$ which we potentially cut again..
- To maintain efficiency again we will memoize the solution using $memo[1..n]$ with every entry initialized to `None`.

```
# Rod Cutting Problem
memo = [ None for x in range(10000) ] # infinity

# Input: An array p representing the rod, n is its length
# Output: The maximum possible revenue from cutting the rod
def CutRod( p, n ):
    if(memo[n] != None ):
        v = memo[n]                                         # Line 1
    elif( n == 0 ):                                         # Line 2
        v = 0                                               # line 3
    else:                                                   # Line 4
        v = -100000 # negative infinity                     # Line 5
        # Find the maximum revenue for each possible cut
        for i in range( n ):                                # Line 6
            v = max( v, p[i] + CutRod(p, n-i) )             # Line 7
    memo[n] = v                                             # Line 8
    return v                                                # Line 9
```

- **Solution**: The solution to the problem is given by CutRod(p, n).
- I claim that this algorithm runs in polynomial time.

- Specifically, $\Theta\left(n^2\right)$ where n is the length of the rod.

- We make the following observations:
  - In the worst case we solve all n possible subproblems.
  - Subproblem j requires us to perform j operations (ignoring recursion costs).
    - Namely, to run the maximum calculation.
  - We will assume all lookups will be constant.
  - Formally, our running time $T(n)$ is given as:

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n} i \\
&= \frac{n(n+1)}{2} && \text{(Closed form of an arithmetic series.)} \\
&\in \Theta\left(n^2\right).
\end{aligned}
$$

11.3.3 Longest Increasing Subsequence

- Let's turn our attention to a problem called the *Longest Increasing Subsequence* (LIS) problem.
- We can define the problem as follows:

  Definition 3 (LIS Problem)**.**
  ***Input**: Given a sequence S of length n.*
  ***Output**: The length of the longest increasing subsequence of S.*

  - A subsequence of S need not be consecutive elements of the sequence but, must respect relative ordering.
    - e.g., if $S = \langle 3, 5, 7, 1, 9 \rangle$ the sequence $\langle 3, 5, 7, 9 \rangle$ is a valid subsequence.
- Let's apply our steps to arrive at a Dynamic Program to solve this problem.
- **Define Subproblems**: Denote by $L(i)$ the length of the longest increasing subsequence that *ends* with element s_i.
- **Guess at Solutions**: What element will make the subsequence ending at the current element the longest?
- **Relate Subproblems**: The LIS for $L(i)$ given by the the maximum $L(j)$ such that element s_j is less than element s_i.
  - Notice we will look at all possible elements j less than i .
  - The recurrence for $L(i)$ is:

$$L(i) = \begin{cases} \max_{j<i} \{1 + L(j)\} & \text{If } s_j \leq s_i \\ 1 & \text{Otherwise.} \end{cases}$$

    - In words, the LIS ending at s_i is given by the LIS ending at s_j.
- To maintain efficiency again we will memoize the solution using *memo* $[1..n]$ with every entry initialized to `None`.

```
# Longest Increasing Subsequence
memo = [ None for x in range(10000) ] # infinity

# Input: A sequence S of length n, the current index from which
#   we are measuring the increasing sequence
# Output: The length of the longest increasing subsequence of S
def LIS( S, i ):
    if( memo[i] != None ):                              # Line 1
        v = memo[i]                                     # Line 2
    elif( i == 1 ):                                     # Line 3
        v = 1                                           # Line 4
    else:                                               # Line 5
        v = -100000 #-infinity                          # Line 6
        for j in range( i ):                            # Line 7
            if( S[j] <= S[i] ):                         # Line 8
                v = max( v, 1 + LIS(S, j) )             # Line 9
        # if v is still -infinity                       # Line 10
        if( v == -100000 ):                             # Line 11
            v = 1                                       # Line 12
    memo[i] = v                                         # Line 13
    return v                                            # Line 14
```

- **Solution**: The solution to the original question is given by $\max_{1 \leq i \leq n} \{L(i)\}$.

- I claim that this algorithm runs in polynomial time.
  - Specifically, $\Theta\left(n^2\right)$ where n is the length of the sequence.
- We make the following observations:
  - In the worst case we solve all n possible subproblems.
  - Subproblem i requires us to perform i operations (ignoring recursion costs).
    * Namely, to run the maximum calculation.
  - We will assume all lookups will be constant.
  - Formally, our running time $T(n)$ is given as:

$$\begin{aligned}
T(n) &= \sum_{i=1}^{n} i \\
&= \frac{n(n+1)}{2} && \text{(Closed form of an arithmetic series.)} \\
&\in \Theta\left(n^2\right).
\end{aligned}$$

11.3.4 Sub-problem Graphs

- When working with dynamic programming our subproblems relate to each other.
  - These relationships can be graphed as a *subproblem graph*.
- A subproblem graph is a graph $G = (V, E)$ where:
  - V is the set of subproblems – one vertex per subproblem.
  - E is a set of edges where an edge $(u, v) \in E$ if determining an optimal solution to subproblem u depends on an optimal solution to subproblem v.
- We can think of a dynamic program with memoization as a depth-first search of the subproblem graph.
- Subproblem graphs aid in determining running time for dynamic programs.
  - $|V|$ tells us the number of subproblems.
  - The time to compute a solution to a subproblem is proportional to the out-degree of the subproblem's vertex.
  - You can think of the total running time of a dynamic program as linear in the number of vertices and edges.

11.4 But Recursion Can Be Expensive...

- For those of you that want to avoid recursion we have a bottom-up approach.
  - This is used if you want to avoid function call overhead which may be extensive when looking at problems with a large number of subproblems to solve.
- In the *bottom-up* approach we use a table that we fill in order of subproblem dependency.
  - i.e., perform a topological sort of the subproblem graph.
  - You can also think of this as sorting the subproblems by size – solve the smaller subproblem first.
- For a quick example of this approach let's consider how we would computing the n^{th} Fibonacci number using a bottom-up algorithm.

```
# Non-recursive memoized algorithms
def FibonacciMomoizedNonRec( n ):
    F = [ None for x in range(n) ]             # Line 1
    for i in range(1,n):                       # Line 2
        if( i <= 2 ):                          # Line 3
            F[i] = i                           # Line 4
        else:                                  # Line 5
            F[i] = F[i-1] + F[i-2]             # Line 6
    return F[n]                                # Line 7
```

  - Notice that from an analysis standpoint it is very clear that this is a $\Theta(n)$ algorithm.
  - In general the bottom-up algorithm is *slightly* easier to analyze.
    - Just be careful how you fill the table.
- We can repeat this with the rod cutting problem as well.

```
#Non-recursive memoized rod cutting
def CutRodNonRec( p ):
    n = len(p)                                 # Line 1
    r = [ None for x in range(n) ]             # Line 2
    for j in range(1, n):                      # Line 3
        v = -100000 #-infinity                 # Line 4
        for i in range( 1, j ):                # Line 5
            v = max( v, p[i] + r[j-i] )        # Line 6
        r[j] = v                               # Line 7
    return r[n]                                # Line 8
```

- Again notice the analysis is simple, we get $\Theta\left(n^2\right)$.
- Something else you might notice is that there appears to be an almost automatic transformation at work.
  - You would be right.
  - If you have a top-down memoized recursive solution it is *almost* trivial to go to a bottom-up approach.
    - The loops are used to "unroll" the recursion.

11.4.1 Single-Source Shortest Path

- Let's look at the original dynamic programming problem (due to Bellman).
- The idea is to find the shortest path in a graph from a given source vertex to all other vertices in the graph
- Applications
  - transportation planning
  - Packet routing in communication networks
  - Friend discovery in social networking
    - Think friend recommendations in Facebook.
  - Speech recognition
- Formally we define the Single-Source Shortest-Paths problem (SSSP) as:

  Definition 4 (Single-Source Shortest-Paths (SSSP) Problem)**.**
  ***Input**: A weighted directed graph $G = (V, E)$ and a source vertex $s \in V$*
  ***Output**: The set of shortest paths*

$$\left\{ p \;\middle|\; s \overset{p}{\rightsquigarrow} v \textit{ is a shortest path from } s \textit{ to } v \textit{ where } v \in V \right\}$$

- This is a combinatorial problem so it appears that dynamic programming will help.
  - It turns out, however, to be nontrivial.
  - Unlike other problem we have seen the *natural* solution won't work.
- Let's get some terminology out of the way. Given a weighted directed graph $G = (V, E)$ we define:
  - We denote the weight of an edge $(u, v) \in E$ as $w(u, v)$.
  - The minimum distance between two vertices $u, v \in V$ as the sum of the weights along a path $u \overset{p}{\rightsquigarrow} v$.
    - p is the sequence of vertices in the path.
    - We denote this as $\delta(u, v)$
  - We say the *out-degree* of a vertex $u \in V$ is the number of edges in E that have u as source.
    - When needed we will denote this as $\deg^+(u)$.
  - We say the *in-degree* of a vertex $u \in V$ is the number of edges in E that have u as a destination.
    - When needed we will denote this as $\deg^-(u)$.
  - We can further define the *degree* of a vertex u as the sum of the in-degree of u and the out-degree of u.
    - We will denote this as $\deg(u)$.
    - We can formally define $\deg(u)$ as:

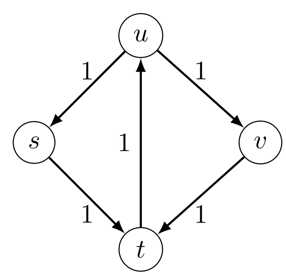
$$\deg(u) = \deg^+(u) + \deg^-(u)\,.$$

- We will relax the problem some. For brevity we will only compute the weight of the minimal weight path.
  - I claim it is easy to recover the actual shortest path if I have a working dynamic program. Do you agree? Why?

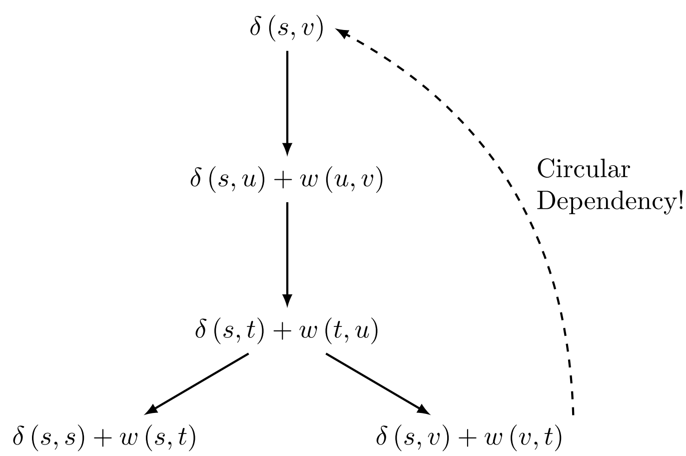
- Let's look at that failed dynamic program I was telling you about.
  - **Define Subproblems**: Let $\delta(u,v)$ be the weight of the shortest path $u \overset{p}{\rightsquigarrow} v$.
  - **Guess at Solutions**: The shortest path $\delta(u,v)$ can be subdivided into a smaller problem.
    * How could we define $\delta(u,v)$?
      · We know that that if $u \overset{p}{\rightsquigarrow} v$ exists, there must be a some vertex $t \in V$ such that $(t,v) \in E$
      · So we can define $\delta(u,v) = \delta(u,t) + w(t,v)$ for some t.
    * How many t are possible?
      · This is nothing more than $\deg^-(v)$.
      · Technically, we should have $\deg^-(v) + 1$.
  - **Relate Subproblems (Recurrence)**: Let's write our recurrence

$$\delta(s,v) = \begin{cases} 0, & \text{If } s = v \\ \min_{(t,v)\in E} \{\delta(s,t) + w(t,v)\}, & \text{Otherwise.} \end{cases}$$

  - **Solution**: $\{\delta(s,v) \mid v \in V\}$.
  - I'm confident that you can write a top-down or bottom-up dynamic program to solve this problem – we'll skip it.
  - I claim that this dynamic program has infinite running time in the worst case. How?
    * Think about what happens when the graph G has a cycle.
    * Consider the following graph:



    * Let's draw the recursion tree for the call $\delta(s,v)$



    * Notice our recursion tree for a graph with cycles has a circular dependency. This means the subproblem graph is not a DAG and therefore we can't use dynamic programming.
- Can we overcome the problem with cylces?

- If we could find a way to deal with cycles in our graph we could eliminate circular dependencies in our subproblem graph (i.e. make it a DAG).

- As it turns out, there is a way to make a dynamic program if we think about the SSSP problem a bit harder. Consider the following observations about $\delta(s,t)$.
  - If we don't allow negative weight edges, does it make sense for a cycle to be in a shortest path?
    * No! if all edges have non-negative weights than a trip around a cycle will only increase the total weight of the path.
    * **Practice**: Is everyone clear why?
    * A path with no cycles is called a *simple path.*
  - What is the maximum number of edges we could have in a shortest path $s \overset{p}{\rightsquigarrow} t$?
    * $|V|-1$ edges. We know that a p is only the shortest path if it does not contain any cycles. This means that we can visit each vertex at most once which gives us $|V|-1$ edges.

- If we consider the above we can make the following key observation:

  > We construct a dynamic program whose recursion tree is depth limited by the maximum number of edges possible in any maximum length simple path (i.e., $|V|-1$).

- Notationally, lets define by $\delta_k(s,v)$ the minimal weight path $s \overset{p}{\rightsquigarrow} v$ such that $|p| \leq k$.

- We can basically, just tune up our recurrence relation to obtain a working single source shortest path dynamic program.
  - In this case, our recurrence $\delta_k(s,v)$ is given by:

$$\delta_k(s,v) = \begin{cases} 0, & \text{If } k = 0 \text{ or } s = v \\ \min_{(t,v)\in E} \{\delta_{k-1}(s,t) + w(t,v)\}, & \text{If } k > 0 \text{ and } s \neq v. \end{cases}$$

  - The solution to our original problem is given by: $\left\{\delta_{|V|-1}(s,v) \mid v \in V\right\}$.

- We can write up a top-down memoized dynamic program.
  - As always assume that $memo[1..|V|, 1..|V|]$ is a memo pad with all cells initialized to `None`.
  - WLOG assume every $v \in V$ is a unique number between 1 and $|V|$ inclusively. We will need two proceedures one that determines the shortest path s to v (ShortestPath) and one that will call ShortestPath for all possible $v \in V$ (SingleSourceShortestPath).

```python
# Dynamic Programming - shortest path
def ShortestPath( E, V, s, v, k ):
    if( memo[u][v] != None ):                          # Line 1
        v = memo[u][v]                                 # Line 2
    elif( k == 0 ):                                    # Line 3
        return 0                                       # Line 4
    elif( u == v ):                                    # Line 5
        v = 0                                          # Line 6
    elif( k > 0 ):                                     # Line 7
        minimum = 1000000 #infinity                    # Line 8
        for e in E:                                    # Line 9
            v = ShortestPath(E,V,s,y,k-1) + w(e)       # Line 10
            if( v < minimum ):                         # Line 11
                minimum = v                            # Line 12
    memo[u][v] = v                                     # Line 13

def SingleSourceShortestPath( E, V, s ) :
    R  = [ 0 for x in range(len(V)) ]                  # Line 1
    for v in V:                                        # Line 2
        R[v] = ShortestPath( E, V, s, v, len(V)-1 )    # Line 3
    return R                                           # Line 4
```

- Let's analyze the running time of our algorithm.
  - For every vertex t we visit in ShortestPath we perform $\deg^{-}(t)$ work.
    - This gives us, $\sum_{t \in V} \deg^{-}(t) = |E|$.
  - Our algorithm SingleSourceShorestPath makes $|V|$ calls to ShortestPath therefore, the running time of our SSSP dynamic program is: $\Theta(|V|\,|E|)$.
- What we have just looked at is essentially the Bellman-Ford algorithm for SSSP.
  - If you have not seen the Bellman-Ford algorithm it has its own section in the CLRS textbook.

Challenge Problem

Answer the following in Python:

1. Convert the robot coin-collection problem from this unit to be a bottom-up dynamic program, as opposed to the solution in the lecture notes which is top-down.
2. Design a dynamic program to solve the Knight's Tour problem. Use a table, not a graph (so don't use one of the algorithms we talked about in this unit). What are the space and time complexities of your solution?
3. Design a dynamic program to find the length of the longest palindrome subsequence in a sequence of characters.