HOMEWORK 3

Charles Haithcock, `cehaith2`

CSC 505 - Design and Analysis of Algorithms
Steffen Heber

Due: 24 March 2017

Homework should be submitted using WolfWare Submit Admin in PDF, or plain text. To avoid reduced marks, please submit **word/latex-formated PDF file, NOT scanned writing in pdf format**. Scanned writing is hard to read, takes longer to grade, and produces gigantic files. **To simplify grading, please make sure that each problem starts on a new page**. All assignments are due on 9 PM of the due date. Late submission will result in 10%/40% point reduction on the first/second day after the due date. No credit will be given to submission that are two or more days late. Please try out Submit Admin well before the due date to make sure that it works for you.

All assignments for this course are intended to be individual work. Turning in an assignment which is not your own work is cheating. The Internet is not an allowed resource! Copying of text, code or other content from the Internet (or other sources) is plagiarism. Any tool/resource must be approved in advance by the instructor and identified and acknowledged clearly in any work turned in, anything else is plagiarism.

General instruction about how to "give/describe/..." an algorithm, taken from Erik Demaine. **Try to be concise, correct, and complete.** To avoid deductions, you should provide (1) a textual description of the algorithm, and, if helpful, pseudocode; (2) at least one worked example or diagram to illustrate how your algorithm works; (3) a proof (or other indication) of the correctness of the algorithm; and (4) an analysis of the time complexity (and, if relevant, the space complexity) of the algorithm. Remember that, above all else, your goal is to communicate. If a grader cannot understand your solution, they cannot give you appropriate credit for it.

Question 1 (4 pts)

Consider the coin-change problem from homework 2. Given a set of arbitrary denominations $C = (c_1, ..., c_d)$, describe an algorithm that uses dynamic programming to compute the minimum number of coins required for making change. You may assume that C contains 1 cent.

Purpose e) reinforce your understanding of dynamic programming and the coin-change problem.

Textual description of the algorithm with pseudocode

```
def make_change(value, denominations):

    # Table for subproblem calculation of the form (value, max denom)
    C = [(0, 0)]  # base case

    # Bottom up approach, so begin precomputing for all possible values
    for subproblem in xrange(1, value):
        minimum = sys.maxint

        # Begin walking possible denominations
        for denomination in denominations:

            # If current denom is smaller than current subproblem and a lookup
            # causes a recalculation of min amount of required coins...
            if (denomination <= subproblem
                and 1 + C[subproblem - denomination][0] <= minimum):

                # Update the min coins required and grab current denomination
                minimum = 1 + C[subproblem - denomination]
                max_denom = denomination

        # Walked all possible denoms, so record min coins and max denom for
        # current subprob
        C.append((minimum, max_denom))

    # Precomputed all possible subprobs, so report min coins and walk table
    # backwards to make change
    print("Minimum coins: " + C[value][0])
    print("Coins back: ", end='')
    while value > 0:
        print(str(C[value][1]) + ", ", end='')
        value = value - C[value][1]
```

Consider the above algorithm where `denominations` is a list of denominations assumed to be pre-sorted in ascending order and `value` is the amount of money to make change for.

For a dynamic programming approach, solutions to subproblems must be tabulated. Subproblems then are intermediate values, $0 \leq x \leq$ `value`. As is typical for dynamic programming approaches, a bottom-up approach is taken here where the tabulated values (represented as `C` in the code above) contain the minimum number of coins required to represent a value at `C[value]` as well as the largest denomination usable for that value.

To build up the table, `C` is created with a base case, `value = 0`. From here, the possible values from 1 to `value` are iterated over to calculate the optimal solution for each subproblem.

For each subproblem, the denominations are walked from lowest value to highest. For the current denomination, if the subproblem can use the denomination then we need to consider the denomination. From here, assume the one coin of the current denomination is selected counting towards the minimum amount of coins for the current subproblem. Take this denomination's value from the subproblem and look up the minimum amount of coins required for the resulting subproblem. If the choice of the current denomination summed with the resulting subproblem's minimum amount of coins is less than our current running minimum amount of coins for the current subproblem, then update our minimum. Once we have walked all possible denominations less than the current subproblem can use, then our current minimum coin count should be the minimum amount of coins required for the current subproblem. Record it.

Once the table is built, start from the end of the table and count the amount of each denomination required for representing `value`.

A worked example

Assume `denominations = [1, 3, 5]` and `value = 6`.

Starting with the base case, `C = [(0,0)]`. On the next page is a table representing the iterations. Each row represents a single iteration of the loop denoted by line 9 while subrows are single iterations of the loop denoted by line 14.

Subp.	Min.	Denom.	Loop Work
1	∞	1	$1 \leq 1$ && $1 + C[1-1] = 1 \leq \infty \implies$ `minimum` $= 1$
1	1	3	$3 \not\leq 1 \implies$ `C = [(0,0), (1,1)]`
2	∞	1	$1 \leq 2$ && $1 + C[2-1] = 2 \leq \infty \implies$ `minimum` $= 2$
2	2	3	$3 \not\leq 2 \implies$ `C = [(0,0), (1,1), (2,1)]`
3	∞	1	$1 \leq 3$ && $1 + C[3-1] = 3 \leq \infty \implies$ `minimum` $= 3$
3	3	3	$3 \leq 3$ && $1 + C[3-3] = 1 \leq 3 \implies$ `minimum` $= 1$
3	1	5	$5 \not\leq 3 \implies$ `C = [(0,0), (1,1), (2,1), (1,3)]`
4	∞	1	$1 \leq 4$ && $1 + C[4-1] = 2 \leq \infty \implies$ `minimum` $= 2$
4	2	3	$3 \leq 4$ && $1 + C[4-3] = 2 \leq 2 \implies$ `minimum` $= 2$
4	2	5	$5 \not\leq 4 \implies$ `C = [(0,0), (1,1), (2,1), (1,3), (2,3)]`
5	∞	1	$1 \leq 5$ && $1 + C[5-1] = 3 \leq \infty \implies$ `minimum` $= 3$
5	3	3	$3 \leq 5$ && $1 + C[5-3] = 3 \leq 3 \implies$ `minimum` $= 3$
5	3	5	$5 \leq 5$ && $1 + C[5-5] = 1 \leq 3 \implies$ `minimum` $= 1$
5	1	5	`C = [(0,0), (1,1), (2,1), (1,3), (2,3), (1,5)]`
6	∞	1	$1 \leq 6$ && $1 + C[6-1] = 2 \leq \infty \implies$ `minimum` $= 2$
6	2	3	$3 \leq 6$ && $1 + C[6-3] = 2 \leq 2 \implies$ `minimum` $= 2$
6	2	5	$5 \leq 6$ && $1 + C[6-5] = 2 \leq 2 \implies$ `minimum` $= 2$
6	2	5	`C = [(0,0), (1,1), (2,1), (1,3), (2,3), (1,5), (2,5)]`

Output:

```
Minimum coins:  2

Coins back:  5, 1,
```

Likewise, if the input was the same set of denominations but `value` $= 17$, the output would be:

```
Minimum coins:  5

Coins back:  5, 5, 5, 1, 1,
```

Proof

Proof by induction. `C[subproblem]` is the minimum amount of coins required to make change for `subproblem`. For the optimal solution for `subproblem` > 0, there must exist some coin from the denominations such that `denominations[i]` $\leq$ `subproblem`. From this, the remaining value, `subproblem` $-$ `denominations[i]` $=$ `subproblem'` must also have an optimal solution exist at `C[subproblem']`. To formalize the claim:

$$\texttt{C[subproblem]} = \begin{cases} 0 & \texttt{subproblem} = 0 \\ \max_{i:\texttt{denom[i]} \leq \texttt{subp}} \{1 + \texttt{C[subp - denom[i]]}\} & \texttt{subproblem} > 0 \end{cases} \quad (1)$$

Note `subproblem` and `demoninations` above are shortened to `subp` and `denom` respectively for legibility.

Base Case For proofs via induction, a base case and inductive case need to be shown. For this algorithm, the base case is when `subproblem` or `subproblem'` $= 0$. Inherently, the minimum amount of coins for 0 is 0 and no change is required to be made. Line 4 above enables this.

Inductive case Assume the algorithm above correctly implements the claim and `subproblem > 0` for the inductive case. The loop at lines 11-20 walk the denominations attempting to find the maximum denomination usable for some arbitrary subproblem. Upon finding the maximum denomination possible, a choice of selecting a single coin of the current denomination is made. A lookup is made to a prior calculated subproblem of `subproblem - max_denom` or `subproblem'` to see how many coins represent the result of selecting the maximum denomination at least once. Note a lookup always occurs to prior calculated subproblems. Because of this, the optimal solution is used for `subproblem'` so selecting a single coin to represent `subproblem' + denomination = subproblem` means no other possible optimal solution exists with fewer coins for `subproblem`. Lines 15-19 calculate the optimal amount of coins and line 24 tabulates the optimal amount of coins for the i^{th} subproblem.

Time complexity analysis

All assignment and arithmetic operations are assumed to be constant time, $\Theta(1)$. As such, each line individually incurs a minimum runtime of $\Theta(1)$.

Lines 7-24 are ran for at least `value` times; as such, line 7 incurs at least $\Theta(n)$ computations while lines 8-24 incurs at minimum $\Theta(n-1)$ computations each.

Lines 11-20 are ran at minimum `denominations.length` times as well; as such, line 11 is performed at least $\Theta(m)$ times while lines 15 - 20 are performed at least $\Theta(m-1)$ times. Since this loop is nested inside of the loop denoted at lines 7 - 24, the running time of this loop is $\Theta(nm)$ which dominates the running time of the algorithm.

While hypothetically, the count of unique denominations could approach n for arbitrarily large values of n causing the running time to become $\Theta(n^2)$, in reality, the count of unique denominations is kept small as creating too many denominations for a country's monetary system would create massive issues all around. As such, the count of unique denominations is assumed to be $\lll n$ for arbitrarily large values of n. Furthermore, as n does become become arbitrarily large compared to m, m could be considered a constant value and drop the running time complexity to something closer to $\Theta(n)$.

Question 2 (14 pts total)

a. (12 pts.) Implement a recursive, a dynamic programming, and a memoized version of the algorithm for solving the matrix-chain multiplication problem described in our textbook (Chapter 15), and design suitable inputs for comparing the run times, the number of recursive calls, and the number of scalar multiplications for all three algorithms. Describe the input data (tell us what inputs you used and why these inputs are suitable for the desired measurments), tabulate and plot your measurements and describe and comment your results. Please submit your programs in three separate files: `h3p2_recursive_uid.ext`, `h3p2_dp_uid.ext`, and `h3p2_memoized_uid.ext`, where uid is your unity id and ext is the extension appropriate for your chosen programming language, e.g., cpp for C++, java for Java, py for python, etc. Your report with graphs and comments and a short description of how to run your program (on a VCL Linux machine) should be submitted in a file called `h3p2_uid.pdf`.

b. (2 pts.) Find an optimal parenthesization of a matrix-chain product whose sequence of dimension is $< 5, 2, 4, 7, 3, 9, 7, 8, 6, 3, 7, 5, 5 >$. How many multiplications does this parenthesization require? How many multiplications are required for a parenthesization that multiplies the input matrices in their input order?

Purpose Reinforce your understanding of dynamic programming and the matrix-chain multiplication problem, and practice run time measurements.

Run the code In order to run the code, start your python interpreter:

```
$ python
```

From here, import the module,

```
>>> import h3p2_dp_cehaith2
```

```
>>> import h3p2_recursive_cehaith2
```

```
>>> import h3p2_memoized_cehaith2
```

Then execute the function with an input:

```
>>> h3p2_dp_cehaith2.matrix_chain_order(<LIST>)
```

```
>>> import h3p2_recursive_cehaith2.recursive_matrix_chain(<LIST>, 1, <n>)
```

```
>>> import h3p2_memoized_cehaith2.memoized_matrix_chain(<LIST>)
```

Overview In terms of time, the Dynamic Programming approach is far superior to the other two implementations. In fact, the Recursive implementation would not complete in a reasonable amount of time for even medium length input. Furthermore, the Memoized implementation would not complete for larger inputs due to literal stack overflows, meaning, while Dynamic Programming may not be the most space efficient implementation, it will actually be able to complete in a reasonable amount of time given the system in general is configured to hold the entirety of the matrix in question.

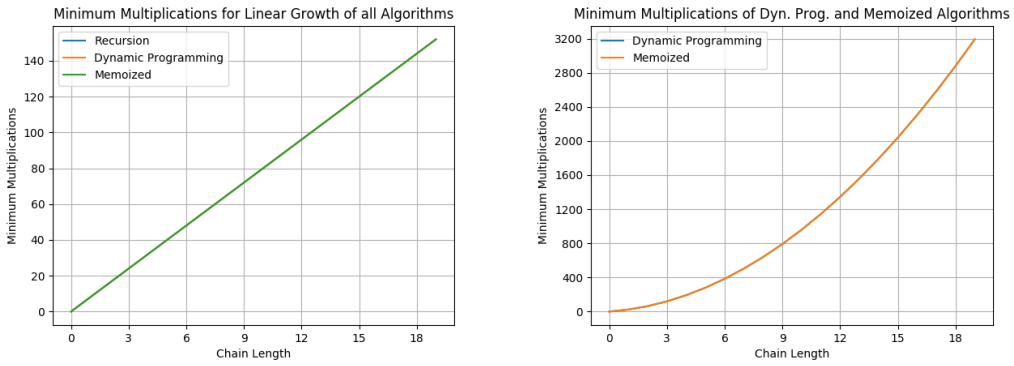
Build The tests were ran on a system with the following specifications:

- CPU: Intel i3-7100U, Quad-core, 2.5 GHz
- RAM: 8 GiB

Input The algorithms' space and time complexity are independent of the actual values of the dimension array and only change based on the length of the dimension array. Because of this the majority of testing for time requirements were performed on basic dimension lists of varying lengths but containing two or more 2's (IE, $\{2, 2, 2, \ldots, 2\}$). To test for correctness in implementation, some simple tests were done on smaller inputs of some varying sizes. Further details herein can be found in the `testing.py` script.

In general, two separate forms of input were tested to observe differences in the algorithms: a series of dimension lists (again, filled with only 2's) which grew linearly from 2 to 20 by an additional length of 1 each iteration, and a series of dimension lists which grew quadratically from 2 to 400 which grew quadratically by an order of 2 (IE $1^2, 2^2, 3^2, \ldots, 20^2\}$. 20 was chosen because, during initial testing to see general completion times, the recursive implementation quickly took an exorbitant amount of time after inputs of length 20 and the memoized implementation would fail on inputs larger than 20^2 due to stack overflow. Note the recursive implementation is left out of testing for recursive input growth due to the time limitations.

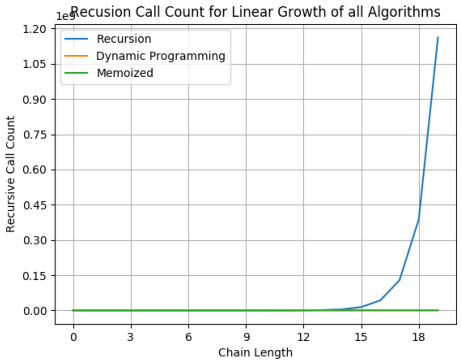
Findings Below are the minimum amount of multiplications for progressively larger inputs. The left figure represents the linear growth described above while the right represents the quadratic grwoth as described above. Note the x-axis for all the quadratic growth plots is the order of the term (tick 18 means 2^{18} for example).

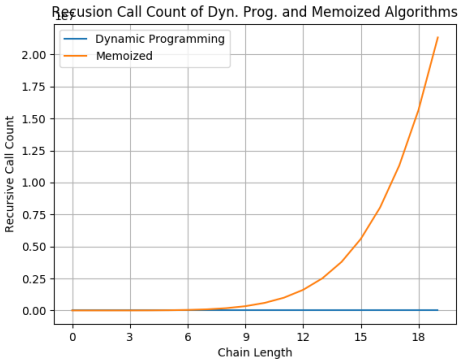


As expected, the minimum amount of multiplications grew linearly with linear growth in input while the amount grew quadratically with quadratic growth in input when the dimensions are the same for all matrices.

Next is the amount of recursive calls. The left image represents the amount of recursive calls with linear growth of input while the right represents quadratic growth of input. Note the y-axis of the left graph below is in terms of 1e9 (IE the 1.05 tick is really 1.05e9 or approximately 1,050,000,000 calls) while the right graph is in terms of 1e7.

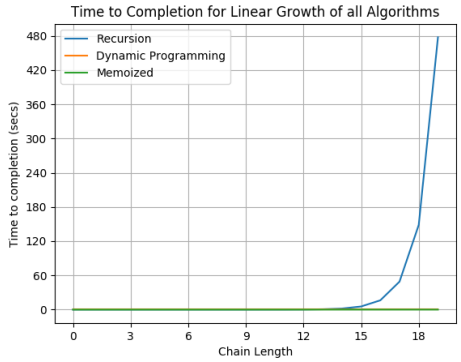
As evidenced above, the recursive implementation makes a staggeringly larger number of

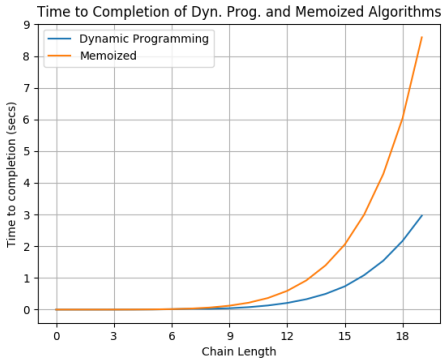




recursive calls than the memoized implementation. Of course, the dynamic programming implementation has the fewest calls because it makes no recursive calls.

And finally, below is the comparison of time to completion. The y-axis for both is in terms of seconds.





As noted earlier, the recursive implementation took an extremely large amount of time on even medium sized dimension lists. Indeed, both the memoized and dynamic programming implementations took very little time on all small inputs compared to the recursive implementation. When looking at only the memoized and dynamic programming approaches, dynamic programming actually performs in logarithmic time compared to the memoized approach on larger inputs.

Note space complexity was not measured in any testing or implementation. If considered, and if the memoized version could be condensed, the memoized version may allow for a great medium between the minimal amount of space required for the recursive implementation and the timing of the dynamic programming approach. This, however, does not take into consideration the amount of space required just for the userspace stack for execution, in which case one could argue the stack space alone, if left unbound, could exceed the space requirements for the dynamic programming approach.

Overall, however, if space is not an issue, a dynamic programming approach is absolutely the best fitting algorithm of the three listed.

The minimum possible multiplications for the described dimension chain is 734. If multiplied in order, the amount of required multiplications is 1750.

Question 3 (6 pts)

Let $G = (U, V, E)$ be a bipartite graph, i.e., a graph where the set of vertices is partitioned into disjoint sets U and V, and every edge (u, v) in E has u in U and v in V. Let $|V| < |U|$ and let vertices in U be numbered $1, \ldots, |U|$ and vertices in V be numbered $1, \ldots, |V|$. The objective is to create a drawing of G in which

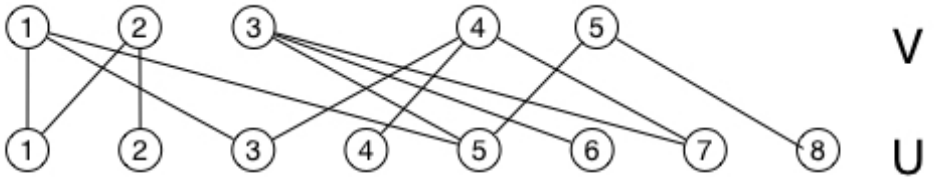
(i) the vertices of U are mapped to integer coordinates $1, \ldots, |U|$ on a horizontal line, no two vertices mapped to the same coordinate;

(ii) the vertices of V are also mapped to coordinates $1, \ldots, |U|$ on a line above the one where the vertices of U appear, and also, no two are mapped to the same coordinate;

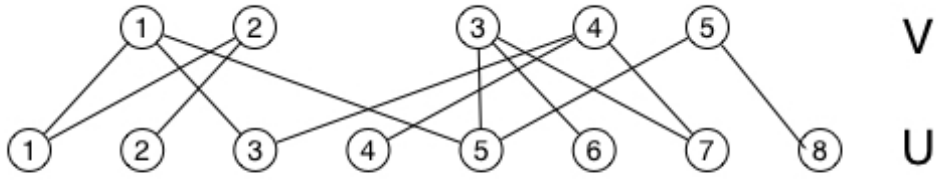
(iii) the vertices of both U and V appear left to right in increasing numerical order; and

(iv) the edges are as "vertical as possible".

To measure of how far away from vertical an edge between i in U and j in V is we define `edgecost(i,j)` to be $(p(j) - i)^2$, where $p(j)$ is the position of vertex j on its horizontal line; note that the positions of the vertices in U are fixed. The picture below shows two different drawings of the same graph G and the computation of total cost (i.e., the sum of all the edge costs) for each; costs for the edges incident on each vertex of V are grouped within parentheses, and the costs of individual edges appear in left to right order. The goal is to minimize total cost.



total cost = (0 + 4 + 16) + (1 + 0) + (4 + 9 + 16) + (4 + 1 + 4) + (1 + 4) = 64



total cost = (1 + 1 + 9) + (4 + 1) + (0 + 4 + 9) + (9 + 4 + 1) + (4 + 1) = 48

If vertex j of V is placed in position k, we define `vcost(j,k)` as the sum of edge costs

of the edges incident on vertex j. In the above example, `vcost(1,1)` $= 0+4+16 = 20$, `vcost(1,2)` $= 1+1+9 = 11$, `vcost(2,2)` $= 1+0 = 1$, `vcost(2,3)` $= 4+1 = 5$, `vcost(2,4)` $= 9+4 = 13$, and `vcost(2,5)` $= 16+9 = 25$.

Devise a dynamic programming algorithm to solve this problem. The runtime of your algorithm should be $\Theta(mn)$ where $m = |U|$ and $n = |V|$. You can assume that `vcost(j,k)` has been precomputed for all relevant j and k and can be retrieved in constant time. Your algorithm should report both the cost of the minimum total cost drawing and the position of each vertex in V in that drawing. **Hint** Let $C[j,k]$ be the minimum total cost of a drawing for vertices $1,\ldots,j$ in V that uses positions $1,\ldots,k$ only. Note that $C[j,k]$ is undefined for $j > k$ and for $k > |U| - (|V| - j)$, and therefore `vcost(j,k)` is not used.

Purpose Practice algorithm design and dynamic programming

Textual description of the algorithm with pseudocode

```
def draw_graph(U, V):

    # Init C to len(V) by len(U) filled with 0s
    C = ([0] * len(U)) * len(V)
    # Used to keep track of which placements create minimum costs
    min_list = [-1]

    # Row wise build up of table
    for j in V:
        jth_min = sys.maxint  # keeps track of current row's minimum cost
        jth_loc = []          # current row's locations for minimizing cost

        # Walk possible placements
        for k in U:

            # Do nothing if out of logical bounds
            if j > k or k > len(U) - (len(V) -j):
                continue

            # Record currently found minimum, updating current minimum
            # where appropriate. Store min cost location as well for
            if C[j - 1][k - 1] + vcost(j, k) < jth_min:
                C[j][k] = C[j - 1][k - 1] + vcost(j, k)
                jth_min = C[j][k]
                jth_loc = k
            else:
                C[j][k] = jth_min

        # Store min cost placement of jth node
        min_list.append(jth_loc)

```

```
    # Clean up the min_list so it has only 1 entry per index. Choose last location
    # in last row and then last possible location elsewhere
    min_list[len(V)] = min_list[len(V)][-1]
    for i in reversed(range(1, len(V) - 1)):
        while min_list[i][-1] >= min_list[i + 1][-1]:
            del min_list[i][-1]
        min_list[i] = min_list[i][-1]

    # Report findings
    print("Minimum Total Cost: " + C[j][k])
    for j, k in enumerate(min_list):
        print("Node: " + j + ", Location: " + k)
```

A dynamic programming approach to the described problems means the subproblems become placement of a subset of V ($v \subset V$) onto a subset of U ($u \subset U$). The computations then become the minimum cost of placing nodes $\{v_1, \ldots, v_j\}$ over possible locations in $\{u_1, \ldots, u_k\}$.

The subproblem precalculation comes into play when attempting to find the minimum costs of node placements. When considering where to place vertex j between locations $\{u_1, \ldots, u_k\}$, the table is referenced to see the minimum cost of placing nodes $\{v_1, \ldots, v_{j-1}\}$ at locations $\{u_1, \ldots, u_{k-1}\}$ summed with `vcost(j, k)`. This summation represents finding the optimal solution for the current subproblem by simulating selection of placing v_j at u_k and calculating the cost of such a placement assuming $\{v_1, \ldots, v_{j-1}\}$ are placed at their optimal locations within $\{u_1, \ldots, u_{k-1}\}$.

For the base case, the table, `C` is instantiated to a zero matrix of size (length of V) x (length of U). The base case is placement of no vertices encountering no cost. Upon completion, the table will have the minimum total cost at the bottom right corner, `C[n][m]` for placement of all nodes. The minimum cost may exist elsewhere on `C[n]` or `C[n - 1]` depending on the results of `vcost(j, k)`.

The algorithm maintains a second data structure to easily keep track of node placement. The i^{th} entry in `min_list` is the locations in row i where the cost shrunk when moving to the next location. This must be maintained in the event the placement choices in a subproblem end up extending past placement choices in a parent subproblem. Such choices would violate the requirement where nodes must be placed in order. When the table building part of the algorithm completes, `min_list` must be cleaned up to select the right most possible locations which minimize costs. Upon termination of cleaning up `min_list`, it will contain indeces satisfying the requirements as well as minimizing costs.

A worked example

$V = \{1, 2\}, U = \{1, 2, 3, 4\}$, edges: $\{(1, 4), (2, 1)\}$.

$$C = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2}$$

For the first subproblem calculation, $C[1, 1]$ causes a lookup to $C[0, 0]$ and adds this to

`vcost(1, 1)`. $C[0,0]$+`vcost(1, 1)` $= 0 + 9 = 9$. Since `jth_min` $= \infty$, `jth_min` is updated to 9 and `jth_loc` $= 1$. This updates C to

$$C = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3}$$

After completing the loop in lines 14-27, `min_list = [[1, 2, 3]]` and C looks like

$$C = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4}$$

Note, $C[1,4]$ is still 0 because vertex v_1 can not be placed at u_4 as that would prevent placement of vertex v_2. This applies similarly to the next row with $C[2,1]$ and vertex v_2 as u_1 as that would prevent placement of v_1.

Upon completion of the table building part of the algorithm, `min_list = [[1, 2, 3], [2, 3]]` and C looks like

$$C = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 4 & 1 & 0 \\ 0 & 0 & 10 & 8 & 8 \end{bmatrix} \tag{5}$$

Note when calculating $C[2,4]$ the value is not 8, but rather 10 as $C[1,3]$+`vcost(2, 4)` $= 10$. This is, however, greater than the minimum value seen up to that point for row 2. As such, we record only the min value observed so far rather than the actual value.

Upon cleaning up `min_list`, the last entry in the end of the list is chosen as a replacement for the last entry and changes the list to `[[1, 2, 3], 3]`. On the next iteration, 3 must be tossed from the first entry as it collides with the already chosen location in the second entry. Once tossed out, we can choose the next index in the entry causing `min_list = [2, 3]`. From here, the output is as follows:

```
Minimum Total Cost:  8

Node:  1, Location:  2

Node:  2, Location:  3
```

Proof

Due to solving subproblems and storing the computed solutions, we can prove the algorithm via induction. The base case herein would indicate whenever the base case of the algorithm is encountered, the result is an immediate lookup and therefore $\Theta(1)$. Because the algorithm's implementation works to build up the solutions from the "bottom up" (IE start at the bottom of the recursion tree implied through the algorithm up through the parent subproblems), computations are always simple lookups and a constant amount of multiplications. Because the subproblems will have the optimal solutions, their parent subproblems will use the optimal solutions to build their optimal solutions.

Time complexity analysis

Each line of code is considered to execute at minimum once but requires constant time to run once, so each line is initially assigned a complexity $\Theta(1)$.

Lines 9 - 30 denote the outer loop required to iterate over the rows, designated as the length of V or n. Because this is a loop, the lines will be ran no more or less than n times and causes the running time complexity to grow to $\Theta(n)$.

Lines 14-17 denote an inner loop nested within the above outer loop. The outer loop is conditioned on the length of U or m. Because this loop will run no more or less than m times, the inner loop has a running time of $\Theta(m)$.

Since the inner loop will be ran n times as well, the inner loop and outer loop end up dominating the run time of the algorithm increasing the copmlexity to $\Theta(mn)$.

Question 4 (6 pts)

Suppose you have a long straight country road with houses scattered at various points far away from each other. The residents all want cell phone service to reach their homes and we want to accomplish this by building as few cell phone towers as possible.

More formally, think of points $x_1, \ldots, x_n$, representing the houses, on the real line, and let d be the maximum distance from a cell phone tower that will still allow reasonable reception. The goal is to find points $y_1, \ldots, y_k$ so that, for each i, there is at least one j with $|y_j - x_i| \leq d$ and k is as small as possible.

Describe a greedy algorithm for this problem. If the points are assumed to be sorted in increasing order your algorithm should run in time $O(n)$. Be sure to describe the greedy choice and how it reduces your problem to a smaller instance. Prove that your algorithm is correct.

Purpose Practice designing greedy algorithms.

Textual description of the algorithm with pseudocode

The algorithm takes as input `road`, a bitmask or list where `road[0]` is the first location on the road where a house could (or could not) be and `road[n - 1]` is the last possible location. Where `road[i]` $= 1$, a house exists, and 0 otherwise.

To begin, the algorithm walks `road` to find the first house and records the index. `road` is traversed further until the current location is distance d away from the recorded index of the first house not covered by a tower or the end of the road is encountered. At this point, a tower is placed (the current index into `road` is recorded). The algorithm restarts at the point where it continues to walk to find the first house not covered by a tower. On termination, the list of tower indeces is returned.

```
def plot_towers(road, d):
    towers = []
    first = -1  # Index of first non-covered house

    for i in range(0, len(road)):

        # Mark index of first non-covered house
        if road[i] and first < 0:
            first = i

        # If placing a tower and d steps away from first house or at road end,
        # record index and go back to searching for non-covered house
        if not first < 0 and (abs(i - first) == d or i == len(road) - 1:
            towers.append(i)
            first = -1

    return towers
```

A worked example

Assume `road` $= \{0, 0, 1, 0, 1, 0, 0, 1\}$ and $d = 3$. The algorithm walks starting from index 0 until index 2 is encountered as the first non-covered house. `first` then becomes 2 and the algorithm continues until index 5. At this point, $|i - d| =$ `first` so `towers` $= \{5\}$. From here, `roads` is traversed further for the next non-covered house. However, the list end is reached before another house is found, so $\{5\}$ is returned.

In the above example, the coverage of houses can not be done with fewer towers as fewer towers is 0. As such, the solution is also the optimal solution which minimizes the amount of towers necessary while covering all houses.

Proof

When moving to the next location to place a tower, the possible spots are parsed until the left-most tower will be the left-most tower which can be covered by the current tower in question. Assume some tower exists outside of the previously placed tower's coverage but is before the currently left-most tower possible by the current tower placement under consideration and therefor is not currently covered by the current tower placement choice under consideration. This means a tower is not covered by the tower placement algorithm. However, this is impossible, as part of the algorithm is specifically to place the next tower as far away as possible from the left-most house which is currently not covered as possible while still being able to cover it.

Time complexity analysis

All lines in the pseudocode are constant time operations, $\Theta(1)$. However, line 5 is a loop walking the length of `road` and is therefore executed n times. As such, line 5 through 15 execute n times. Each of the loop body statements are constant time, so the complexity is dominated by the loop executing n times. Furthermore, because the loop will always run n times, complexity is then $\Theta(n)$.

Some considerations for this algorithm include future expansions to the road and representation of the `road` input. For example, if houses will be built, the placements may no longer be optimal. Likewise, if the `road` is a bit mask, `road` can be segmented and each segment be checked to have values above 0. While the asymptotic complexity will not improve from $\Theta(n)$, the amortized analysis could potentially be closer to $\Theta(\log(n))$ for more sparse roads. Heavily populated roads, however, would likely end up devolving into something along the lines of $\Theta(mn)$ where m is the amount of segments.