* 1. Prim’s Algorithm

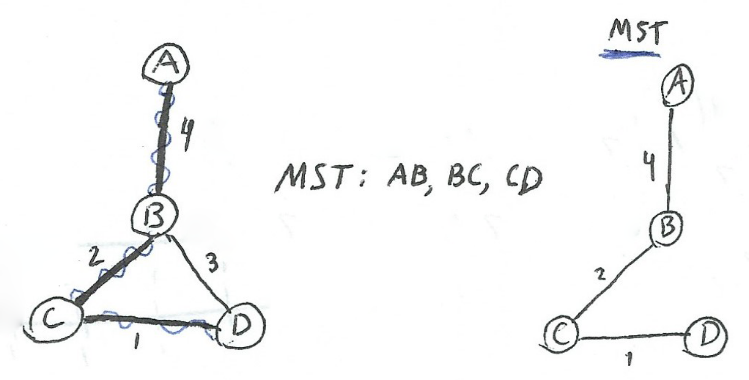
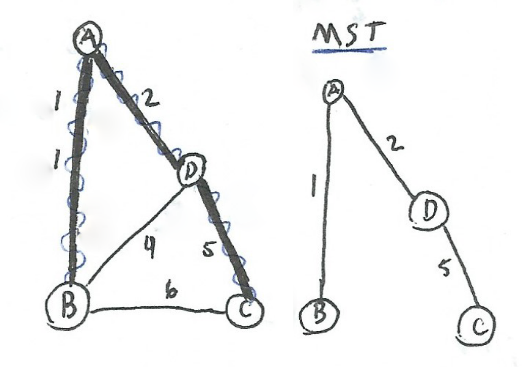
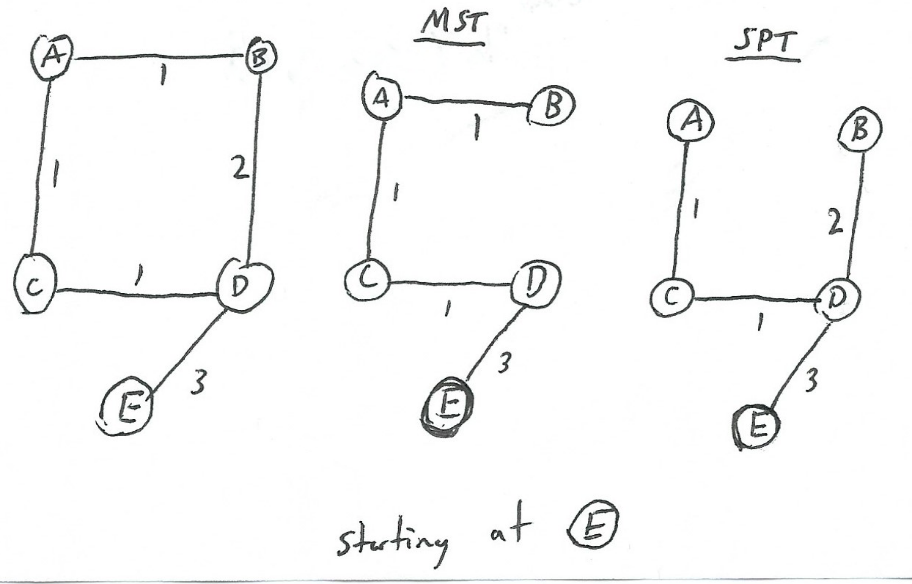
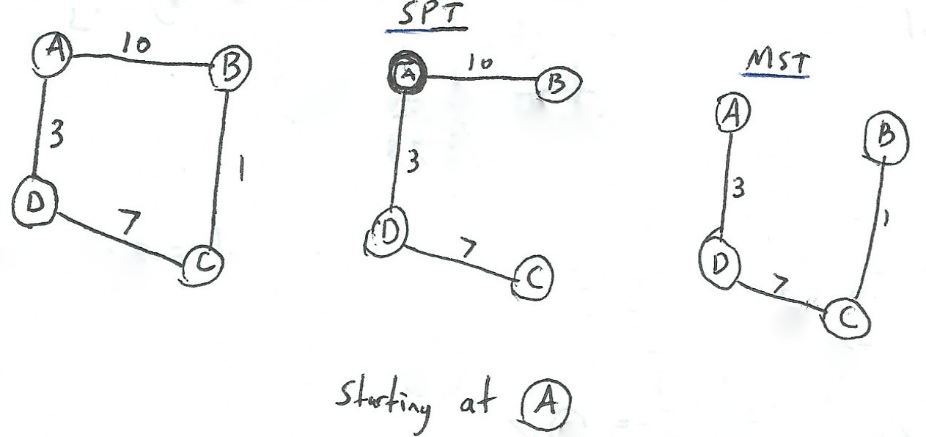
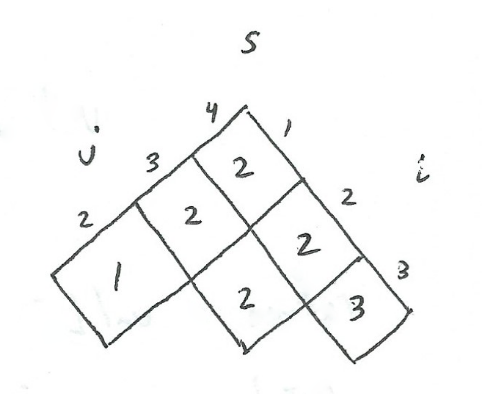
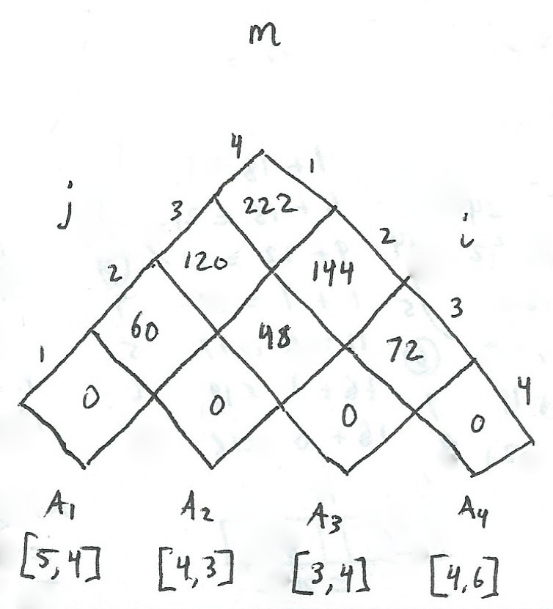
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| <step> | set S | A | B | C | D | E | F |
| Init | { } | *0/nil* | ∞/nil | ∞/ni2l | ∞/nil | ∞/nil | ∞/nil |
| 1 | {A} |  | 8/A | 6/A | ∞/nil | *1/A* | ∞/nil |
| 2 | {A, E} |  | 7/E | *6/A* | 10/E |  | 12/E |
| 3 | {A, C, E} |  | 7/E |  | 10/E |  | *2/C* |
| 4 | {A, C, E, F} |  | *7/E* |  | 8/F |  |  |
| 5 | {A, B, C, E, F} |  |  |  | *3/B* |  |  |
| 6 | {A, B, C, D, E, F} |  |  |  |  |  |  |

* 1. Kruskal’s Algorithm  
     Initialize the sets as: {A} {B} {C} {D} {E} {F}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Weight | Edge | Used | Notes | Set(s) |
| 1 | AE | Y | A union E | {A, E} {B} {C} {D} {F} |
| 2 | CF | Y | C union F | {A, E} {B} {C, F} {D} |
| 3 | BD | Y | B union D | {A, E} {B, D} {C, F} |
| 6 | AC | Y | A union C -> {A, E} union {C, F} | {A, C, E, F} {B, D} |
| 7 | BE | Y | B union E -> {A, C, E, F} union {B, D} | {A, B, C, D, E, F} |
| 7 | CE | N | *Algorithm is complete* |  |
| 8 | AB | N |  |  |
| 8 | DF | N |  |  |
| 10 | DE | N |  |  |
| 12 | EF | N |  |  |

Directed Trees

|  |  |
| --- | --- |
| Init |  |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 | No path compression (up to and including this step) |
| 5 | Path compressed from Find(E) so B connects directly to A |

1. Graph Statements
   1. False, if the heaviest edge is the only edge connecting a given vertex. 
   2. True, removing a cycle edge cannot disconnect a graph.
   3. True, because both algorithms add a minimum edge to their MST first thing. This edge must be part of some MST, but not necessarily every MST since it’s not specified to be the uniquely lightest edge. (More info in 2D on this)
   4. True, this edge is added immediately to the MST on both Kruskal and Prim’s algorithms. It also cannot create a cycle in the case of Kruskal’s since it’s the only edge of the MST being built so far.
   5. True, due to the cut property for MST.
   6. False, if adding that unique lighted edge of a cycle would create a cycle in the MST.  
       
   7. False  
       
   8. False, the shortest path from the SPT starting at A, includes the edge AB. However, such an edge does not appear in the same graph’s MST.  
      
   9. True, it sorts the edges by their weight by increasing order and prevents cycles from being produced.
2. Optimal Matrix Multiplication Order  
     
   The optimal matrix multiplication order for the given matrices is:  
   (A1 A2) (A3 A4)  
   The ordering of these brackets is determined by the s table above. s[1, 4] has the k value 2, and this means the bracket goes after the 2nd matrix and that’s the final step since each set has 2 matrices.

|  |  |
| --- | --- |
| Direction | Bracket |
| → | ( |
| ↑ | ) |

1. Lattice Paths  
   There are as many lattice paths from (0, 0) to (n, n) that do not cross the diagonal as there are well-formed sequences on n opening and n closing brackets.  
     
   I shall map the movement of going right and up on the lattice to open and close brackets respectively.

The first step and last step are both forced, as to not cross the diagonal. The first step is go right and last step is go up. This results in a bracket sequence of ( … ) initially. Now I will prove that going from (1, 0) to (n, n - 1) will produce a well-formed bracket sequence of n - 1 opening and closing brackets each.

For the second bracket there are 2 options:  
- Going up will close the first bracket (making the first set of brackets complete)

Whenever you reach the lattice’s diagonal, all of the brackets already added to the sequence have been closed. This makes your current progress similar to restarting at (0, 0) in that you have only one option of going right and opening a new set of brackets.

- Going right will open another bracket nested in the first one. And this is fine because it will eventually be closed when moving up to the diagonal later and finally at (n, n)

The base case having n = 1 the only option again is right, up equalling ( ).

To help argue my point I will ask “What could create a malformed bracket sequence?”  
Crossing the diagonal means that there has been more closing brackets than opening brackets so far, resulting in a bad sequence. There are also the constraints of only being able to move up and to the right ensuring this proper-sequence. Being confined to the [n x n] lattice also prevents from having more than n of each type of bracket. There can only be n + 1 opening brackets if you have left the original lattice dimensions.

1. Dynamic Programming: Cutting a Rod

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| r[i] | 0 | 1 | 6 | 9 | 12 | 15 | 18 | 21 | 24 |
| s[i] | 0 | 1 | 2 | 3 | 2 | 2 or 3 | 2, 3 or 6 | 2 or 3 | 2, 3 or 6 |

The array s can have multiple cuts that result in the same revenue from the rod, I kept all occurences of this however, realistically only one is needed. Using the greatest of the s[i] values possibly could minimize the number of cuts (which in reality should have a cost associated with them, but did not in the question).  
The best ways to cut a rod of length 8 for the given price structure would be:

* Segments of size 2 and 6 --> 6 + 18 = 24
* Segments of size 2, 2, 2 and 2 --> 4 \* 6 = 24
* Segments of size 2, 3, and 3 --> 2 \* 9 + 6 = 24

The steps taken were to compute the best way to cut the rod for all rods of a smaller size (from 1 to 7). This is done in a bottom-up manner, starting with the best way to cut a rod of size 1, which cannot be but resulting in a value of $1. Then using that optimal value in the next problem for a rod of length 2. This rod can be cut in half or not cut at all, the best way being to leave it at 2 long for $6 rather than two $1 pieces. This sequence is then continued until n = 8 and by storing the s array, we know what locations to cut the rod as well as what it will be worth, r[n]. To get the positions at which to cut the rod optimally, run the following while loop taken from the print-cut\_rod\_solution(p, n) function in the notes:

* 1. While n > 0
  2. print s[n]
  3. n = n – s[n]

So n starts at 8 in this case. Then we have the option of cutting it at 2, 3, or 6. The next cut will depend on this decision, and will make the cut at the remaining size’s optimal cut location in a loop until there’s no rod left.