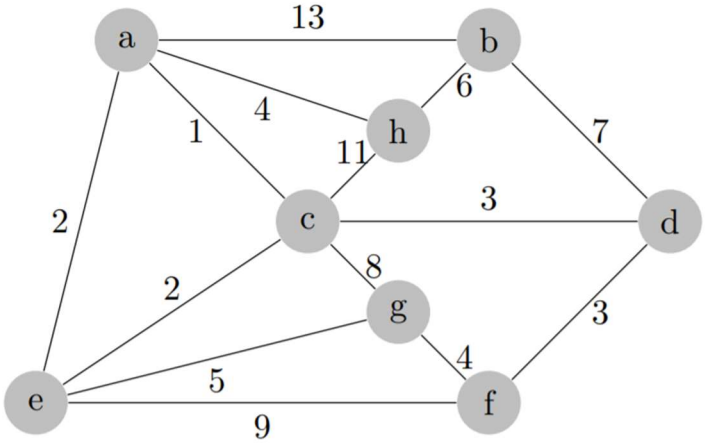
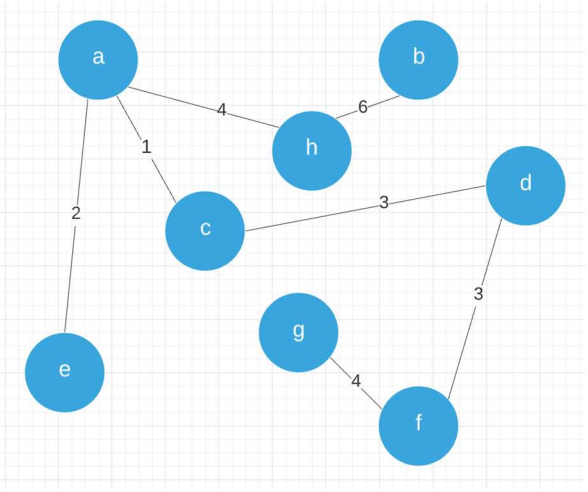
Brandon Vo



1. For the graph above:

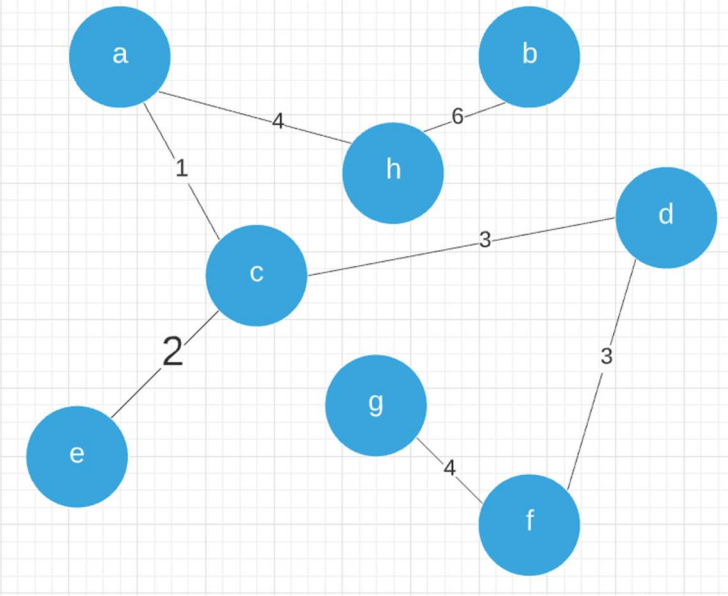
(a) What is the cost of a minimum spanning tree?



- Total Cost = 6 + 4 + 1 + 2 + 3 + 3 + 4 = 23

(b) How many minimum spanning trees does it have?

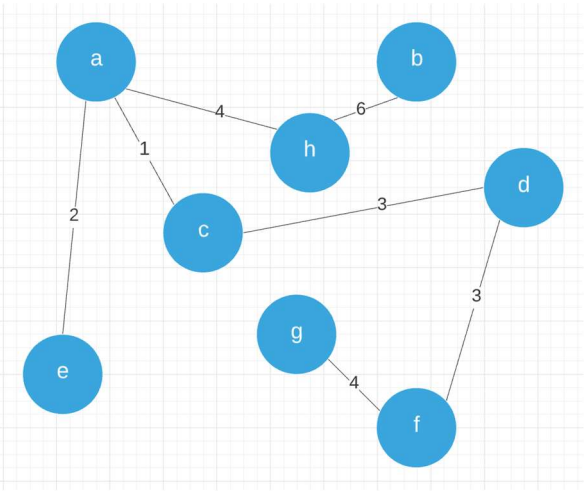
Brandon Vo



- We can replace (a, e) with (c, e) and maintain the same cost of the MST

There are 2 different minimum spanning trees contained in this graph.

(c) For the graph above, run Prim's algorithm. Start at node a. Whenever there is a choice of nodes, always use alphabetic ordering. Show the order the vertex are removed from Q, and draw the minimum spanning tree T for the graph.



Starting at node A:

- (a, c) – 1
- (a, e) – 2
- (e, c) – 2 (Creates a cycle)
- (c, d) - 3
- (d, f) – 3

Brandon Vo

- (f, g) – 4
- (a, h) – 4
- (e, g) – 5 (Creates a cycle)
- (b, h) – 6
- (b, d) – 7 (Creates a cycle)
- (c, g) – 8 (Creates a cycle)
- (e, f) – 9 (Creates a cycle)
- (c, h) – 11 (Creates a cycle)

Brandon Vo

2. (15 points) Suppose Joseph Kruskal had an evil twin, named Peter, who designed an algorithm that takes the exact opposite approach from his brother's algorithm for finding an MST in an undirected, weighted, connected graph, G. Also, for the sake of simplicity, suppose the edges of G have distinct weights. Peter's algorithm is as follows: consider the edges of G by decreasing weights. For each edge, e, in this order, if the removal of e from G would not disconnect G, then remove e from G. Peter claims that the graph that remains at the end of his algorithm is a minimum spanning tree. Prove or disprove Peter's claim.

- Peter's algorithm checks for the highest weight edge and removes it if it's part of a cycle
- If an MST is unique, then all MST algorithms will yield the same minimum spanning tree.
- An MST requires only the minimum edges with no cycles which connects every vertex in the graph

- For each loop of Peter's algorithm, it removes an unnecessary edge with the largest weight.
    - We start with a graph that all edges. If this graph has no cycle, then the MST is already found and nothing needs to be done.
    - If the graph has a cycle, then we remove the edge with the largest weight which preserves one of MST's main properties.
    - Each loop iteration removes the largest edge until there are no more cycles which means we end up with a graph that has no cycle, all vertices remain connected, and only the largest edges have been removed.
    - This means we still end up with an MST, meaning Peter's algorithm is correct.

Brandon Vo

3. What is the best way to multiply a chain of matrices with dimensions that are 15 × 3, 3 × 7, 7 × 27, 27 × 10, 10 × 5, and 5 × 60? Show your work.

0	1	2	3	4	5	6
15	3	7	27	10	5	60

The best way to multiple would be to multiply the matrices that result in a matrix that has the fewest number of dimensions. Multiplications are counted in the form of p q r.

For multiplying matrices, the format would be $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} * p_k * p_j$

- M[1, 2] = M[1, 1] + M[2, 2] + P[0] * P[1] = 15x13 * 3x7 = 15*3*7 = 315
- M[2, 3] = M[2, 2] + M[3, 3] + P[1] * P[2] = 3 x 7 * 7 x 27 = 3 * 7 * 27 = 567
- M[3, 4] = M[3, 3] + M[4, 4] + P[2] * P[3] = 7 x 27 * 27 x 10 = 7 * 27 * 10 = 1890
- M[4, 5] = M[4, 4] + M[5, 5] + P[3] * P[4] = 27 x 10* 10 x 5 = 27 * 10 * 5 = 1350
- M[5, 6] = M[5, 5] + M[6, 6] + P[4] * P[5] = 10 x 5 * 5 x 60 = 10 * 5 * 60 = 3000

- M[1, 3]
  - M[1, 1] + M[2, 3] + P[0] * P[1] * P[3] = 0 + 567 + 15 * 3 * 27 = **1782**
  - M[1, 2] + M[3, 3] + P[0] * P[2] * P[3] = 315 + 0 + 15 * 7 * 27 = 3150
- M[2, 4]
  - M[2, 2] + M[3, 4] + P[1] * P[2] * P[4] = 0 + 1890 + 3 * 7 * 10 = 2100
  - M[2, 3] + M[4, 4] + P[1] * P[3] * P[4] = 567 + 0 + 3 * 27 * 10 = **1377**
- M[3, 5]
  - M[3, 3] + M[4, 5] + P[2] * P[3] * P[5] = 0 + 1350 + 7 * 27 * 5 = 2295
  - M[3, 4] + M[5, 5] + P[2] * P[4] * P[5] = 1890 + 0 + 7 * 10 * 5 = **2240**
- M[4, 6]
  - M[4, 4] + M[5, 6] + P[3] * P[4] * P[6] = 0 + 3000 + 27 * 10 * 60 = 19200
  - M[4, 5] + M[6, 6] + P[3] * P[5] * P[6] = 1350 + 0 + 27 * 5 * 60 = **9450**

- M[1, 4]
  - M[1, 1] + M[2, 4] + P[0] * P[1] * P[4] = 0 + 1377 + 15 * 3 * 10 = **1827**
  - M[1, 2] + M[3, 4] + P[0] * P[2] * P[4] = 315 + 1890 + 15 * 7 * 10 = 3255
  - M[1, 3] + M[4, 4] + P[0] * P[3] * P[4] = 1782 + 0 + 15 * 27 * 10 = 5832
- M[2, 5]
  - M[2, 2] + M[3, 5] + P[1] * P[2] * P[5] = 0 + 2240 + 3 * 7 * 5 = 2345
  - M[2, 3] + M[4, 5] + P[1] * P[3] * P[5] = 567 + 1350 + 3 * 27 * 5 = 2322
  - M[2, 4] + M[5, 5] + P[1] * P[4] * P[5] = 1377 + 0 + 3 * 10 * 5 = **1527**
- M[3, 6]
  - M[3, 3] + M[4, 6] + P[2] * P[3] * P[6] = 0 + 9450 + 7 * 27 * 60 = 20790
  - M[3, 4] + M[5, 6] + P[2] * P[4] * P[6] = 1890 + 3000 + 7 * 10 * 60 = 9090
  - M[3, 5] + M[6, 6] + P[2] * P[5] * P[6] = 2240 + 0 + 7 * 5 * 60 = **4340**

Brandon Vo

- M[1, 5]
  - M[1, 1] + M[2, 5] + P[0] * P[1] * P[5] = 0 + 1527 + 15 * 3 * 5 = **1752**
  - M[1, 2] + M[3, 5] + P[0] * P[2] * P[5] = 315 + 2240 + 15 * 7 * 5 = 3080
  - M[1, 3] + M[4, 5] + P[0] * P[3] * P[5] = 1782 + 1350 + 15 * 27 * 5 = 5157
  - M[1, 4] + M[5, 5] + P[0] * P[4] * P[5] = 1827 + 0 + 15 * 10 * 5 = 2577
- M[2, 6]
  - M[2, 2] + M[3, 6] + P[1] * P[2] * P[6] = 0 + 4340 + 3 * 7 * 60 = 5600
  - M[2, 3] + M[4, 6] + P[1] * P[3] * P[6] = 567 + 9450 + 3 * 27 * 60 = 14877
  - M[2, 4] + M[5, 6] + P[1] * P[4] * P[6] = 1377 + 3000 + 3 * 10 * 60 = 6177
  - M[2, 5] + M[6, 6] + P[1] * P[5] * P[6] = 1527 + 0 + 3 * 5 * 60 = **2427**
- M[1, 6]
  - M[1, 1] + M[2, 6] + P[0] * P[1] * P[6] = 0 + 2427 + 15 * 3 * 60 = **5127**
  - M[1, 2] + M[3, 6] + P[0] * P[1] * P[6] = 315 + 4340 + 15 * 7 * 60 = 10955
  - M[1, 3] + M[4, 6] + P[0] * P[1] * P[6] = 1782 + 9450 + 15 * 27 * 60 = 35532
  - M[1, 4] + M[5, 6] + P[0] * P[1] * P[6] = 1827 + 3000 + 15 * 10 * 60 = 13827
  - M[1, 5] + M[6, 6] + P[0] * P[1] * P[6] = 1752 + 0 + 15 * 5 * 60 = 6252

Finding the optimal substructure:

m	1	2	3	4	5	6
1	0	315	1782	1827	1752	5127
2		0	567	1377	1527	2427
3			0	1890	2240	4340
4				0	1350	9450
5					0	3000
6						0

S table:

m	1	2	3	4	5	6
1	0	1	1	1	1	5
2		0	2	3	4	5
3			0	3	4	5
4				0	4	5
5					0	5
6						0

According to the matrix table, the optimal way to multiply would be

$$15 \times 3 * 3 \times 7 * 7 \times 27 * 27 \times 10 * 10 \times 5 * 5 \times 60$$

$$15 * \Big(\Big(\Big(\big((3\ x\ 7) * (7 * 27)\big) * (27\ x\ 10)\Big) * (10\ x\ 5)\Big) * (5\ x\ 60)\Big)\ with\ a\ cost\ of\ 5127$$

Brandon Vo

4. As stated, in dynamic programming we first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that we do not always need to solve all the subproblems in order to find an optimal solution. She suggests that we can find an optimal solution to the matrix chain multiplication problem by always choosing the matrix A_k at which to split the subproduct $A_iA_{i+1} \cdots A_j$ (by selecting k to minimize the quantity $p_{i-1}p_kp_j$) before solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

- Capulet's solution uses k to split the matrix chain multiplication
- She would be splitting by using m[i, j] = min{$p_{i-1}p_kp_j$}

7 x 3, 3 x 2, 2 x 1, 1 x 9

The matrix chain table if done normally

- M[1, 4] = M[1, 3] + M[4, 4] + p[0] * p[3] * p[4]
- M[1, 4] = 27 + 0 + 7 * 1 * 9 = 90

M	1	2	3	4
1	0	42	27	90
2		0	6	33
3			0	18
4				0

This yields a solution of $\Big(\big((7\ x\ (3\ x\ 2))\ x\ 1\big)\ x\ 9\Big)$

The matrix chain table if done by Capulet's solutions

We choose matrix A_k to split the matrix, then we would count only based off the p values

$$A[i,j] = \min_{i \le k < j} p_{i-1}p_kp_j$$

P	0	1	2	3	4
	7	3	2	1	9

- M[1, 2] = P[0] * P[1] * P[2] = 7 * 3 * 2 = 42
- M[2, 3] = P[1] * P[2] * P[3] = 3 * 2 * 1 = 6
- M[3, 4] = P[2] * P[3] * P[4] = 2 * 1 * 9 = 18

- M[1, 3] = MIN{P[0] * P[1] * P[3], P[0] * P[2] * P[3]}
    - 7 * 3 * 1 = 21

  - **7 * 2 * 1 = 14**
- M[2, 4] = MIN{P[1] * P[2] * P[4], P[1] * P[3] * P[4]}
  - 3 * 2 * 9 = 54
  - **3 * 1 * 9 = 27**
- M[1, 4] = MIN{P[0] * P[1] * P[4], P[0] * P[2] * P[4], P[0] * P[3] * P[4]}
  - 7 * 3 * 9 = 189
  - 7 * 2 * 9 = 126
  - **7 * 1 * 9 = 63**

M	1	2	3	4
1	0	42	14	63
2		0	6	27
3			0	18
4				0

The matrix chain would be $\left(\left(\left((7\ x\ 3)\ x\ 2\ \right)\ 1\right)\ x\ 9\right)$ which is a different solution from the standard greedy algorithm.

Brandon Vo

S	1	2	3	4	5
1	0	1	2	3	**5**
2		0	2	3	4
3			0	3	4
4				0	4
5					0

This yields a solution of $5\ x\left(\left(\left((50\ x\ 7)x\ 6\right)x\ 5\right)x\ 10\right)$

Brandon Vo

5. Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the density of a rod of length i to be p_i/i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \le i \le n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

- Greedy algorithms work by tracing the series of local maximums and using those as a form of path to find the global maximum
- However, not all local maximums lead to the global maximums.

Length	1	2	3
Price	1	8	15
P_i/i	1	4	5

The ways to cut a rod of length 5 using Price:

- $R_n = r_i + r_{n-i}$
- $R_1 = 1$
- R_2
  - $R_1 + R_1 = 2$
  - **$R_2 = 8$**
- $R_3 = \text{Max}\{R_1 + C(2), R_2 + C(1), R_3\}$
  - $R_1 + C(2) = 1 + 8 = 9$
  - $R_2 + C(1) = 8 + 1 = 9$
  - **$R_3 = 15$**
- R_4
  - **$R_1 + C(3) = 1 + 15 = 16$**
  - **$R_2 + C(2) = 8 + 8 = 16$**
  - **$R_3 + C(1) = 15 + 1 = 16$**
- R_5
  - $R_1 + C(4) = 1 + 16 = 17$
  - **$R_2 + C(3) = 8 + 15 = 23$**
  - **$R_3 + C(2) = 15 + 8 = 23$**
- The optimal solution would be (3, 2)

If we ran the rod through the greedy algorithm, we would sort based on price density.

- $R_n = r_i + r_{n-i}$
- $R_1 = 1$
- $R_2 = \text{Max}\{R_1 + R_1, R_2\} = \text{Max}\{1 + 1, 4\} = 4$
- $R_3 = \text{Max}\{R_1 + C(2), R_2 + C(1), R_3\}$
  - **$R_1 + C(2) = 1 + 4 = 5$**

    - **R_2 + C(1) = 4 + 1 = 5**
    - R_3 = 3
- R_4
    - R_1+ C(3) = 1 + 5 = 6
    - **R_2 + C(2) = 4 + 4 = 8**
    - R_3 + C(1) = 4 + 1 = 5
- R_5
    - **R_1 + C(4) = 1 + 8 = 9**
    - **R_2 + C(3) = 4 + 5 = 9**
    - R_3 + C(2) = 5 + 4 = 9
- The optimal way of cutting would be (2, 2, 1)

As shown, the greedy algorithm for just price and price density yields two different solutions.
Depending on what someone wants, the greedy algorithm might not yield the wanted optimal solution.

Brandon Vo

6. Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Your algorithm should not only the value but the actual solution, too.

Provide a recurrence formula that would calculate the answer.

Using your recurrence formula, give a dynamic-programming algorithm to solve this modified problem.

Length	1	2	3
Price	1	7	4

With each iteration, each R_n will be decreased by the fixed cost of c. The revenue would already include the cut cost of cutting c.

- $R_1 = 1$
- R_2
  - $R_1 + R_1 - c = 2 - c$
  - **$R_2 = 7$**
- $R_3 = \text{Max}\{R_1 + C(2), R_2 + C(1), R_3\}$
  - **$R_1 + C(2) = 1 + 7 - c = 8 - c$**
  - $R_3 = 4$
- The length of the rod would be cut as [1, 2] if the cost c was > 4. Otherwise, it would just be [3].

$$m[i,j] = \begin{cases} 0 \; if \; i = j \\ \min\limits_{i \le k < j} \{p_n, r_{n-i} - c_i\} \; if \; i < j \end{cases}$$

CUTTING(p, n, c)

Let r[1 … n]] be a new array

R[0] = 0

For j = 1 to n

R[i] = ∞

For i = 1 to **j**

Q = max(q, p[i] + CUTTING(p, r[j – i] - c) //Just change this to include the fixed cost of cutting

//Every time we look at this the cost of cutting r[j – i], the fixed cost of c is already taken into account

R[j] = q

Return r[n]

Brandon Vo

We only need to add – c to the end because whenever we subtract the cost with a cut, that cost difference is already considered when comparing the next iteration of the rod cutting.

Brandon Vo

7. Consider a road of M miles. The task is to designate lots on which to build houses on the road such that revenue is maximized. The possible sites for house are given by number $x_1 < x_2 < \cdots < x_{n-1} < x_n$, specifying positions in miles of the center of each lot measured from one end of the road. If we place a house at lot i (where the center of the lot is in position x_i), we receive a revenue of $r_i > 0$. There is a restriction that no two houses (as measured by the center of each lot) can be placed within t miles or less than it. As a first step in solving this problem, create the recurrence relation that returns the maximum revenue possible while satisfying that not two houses can be placed within t miles or less. What size table do you need to store the subproblems using big-Oh notation. Create an example of where n = 3, and show how your algorithm works on this example.

$$R_n = \begin{cases} 0 \; if \; k \leq t \\ \max\{r[k-t-1] + r[k], r[k-1]\} \\ t \leq k < n \end{cases}$$

- With a road of m miles, we can take the entire road and try to split it n in a way that maximizes revenue.
    - To remember the minimum distance t, any two houses with distance <= t will have a revenue of 0 to always fail the requirement.
    - We can check if building two houses that are more than t miles apart gives more revenue than just building one house
    - We can then recursively check the leftover m miles to see if we can build more sets of houses.
- We will use a hash table called DST to see which house can be built at mile m where 0 < m < n
    - If DST(H[n]) has no match, then it will be 0 to indicate we receive no revenue because we can't build a house here
    - Otherwise, return the revenue found in R[DST(H[n])]
- We will create a second array that stores the distance each house has to check if we have enough distance > t miles.
- Because we're using two arrays that are of size n, we would need O(N) size complexity to use this algorithm.

In a scenario where n = 3 and t = 5 and we're presented with the houses:

DST[n]:

X_n (Location)	H(3)	H(4)	H(10)
N (House #)	1	2	3

With the revenue being saved as R[n]

N	1	2	3

Brandon Vo

R_n (Revenue)	5	2	15

The algorithm will work similarly to the rod-cutting algorithm where we take m miles and see how much we can cut it. If we use m = 10:

- R[10] = MAX{R[4] + X(10), R[9]}
- Must recursively check R where the distance m has been cut
    - R[4] = MAX{R[4 – 6] + X(4), R[3]}
    - R[4 – 6] <= t, so R[4 – 6] = 0
    - X[10] = 15
    - **R[10] = MAX{5 + 15, 4} = 20**
- R[9] = MAX{R[3] + 0, R[8]} = 5
    - R[3] = 5
    - R[9] = MAX{5 + 0, 5}
- R[8] = MAX{R[2] + 0, R[7]} = 5
- R[7] = MAX{R[1] + 0, R[6]} = 5
- R[6] = 0 + 0, R[5]} = 5
- R[5] = MAX{0 + 0, R[4]} = 2
- R[4] = MAX{0 + X[3], R[3]} = 5
    - X[3] = 2
    - R[3] = 5
- R[3] = MAX{R[3 – 5 – 1] + X[3], R[2]}
    - R[3 – 5 – 1] <= t, so R[3 – 5 – 1] = 0
    - X[3] = 5
    - R[2] = 0
    - R[3] = MAX{ 0 + 5, 0} = 5
- R[2] = MAX{0 + R[2], R[1]} = 0
    - R[1] = 0
    - R[2] = 0

```
//Initially, n = m

//DST is a hash table that holds the location of a house at mile mi

//R is an array that holds the revenue for a particular house if built

MEMOIZED-HOUSING(n, DST, R, t)

If (all houses have been placed)

   Return R

If DST[H(n)] has no match          //No hash table collision means no house can be build at mile n

   REV = 0          //Rev returns the revenue we would get if we built the house at mile n
```

Else REV = R[DST[H(n)]] //Record the revenue earned if we built a house at mile m

q = -∞

//MEMOIZED-HOUSING(r[n – t – 1], DST, R, t) + REV means search for the revenue we would earn if we built a house here and another house somewhere before n – t

//MEMOIZED-HOUSING((n-1), DST, R, t)) means search for the revenue if we just built a house at mile n-1

Q = max(q, MEMOIZED-HOUSING((n – t – 1), DST, R, t) + REV, MEMOIZED-HOUSING((n-1), DST, R, t))

R[n] = Q

Return q

Brandon Vo

8. On Friday night, Zorad was in his basement - strategizing about the latest hit strategy game. A light went off in his mind - illuminating the dark basement as he discovered a flaw in the game's conquest mode: if he left an invisible scout unit in the center of a city, the city would become effectively unconquerable until very late in the game.

He realized he could use this fact to seize a massive early-game advantage.

He then mapped out the game map - notating the minimum cost of maintaining his supplies from a conquered city that was sufficient to conquer the adjacent city's initial forces.

Your task is to help Zorad quickly decide the order of the cities he should conquer at the minimum cost (you must say which road he used by stating the two cities). Zorad has already conquered one city - the city of Algorithmica. In addition your algorithm must be efficient to account for the actions of other players which might change the cost of taking over a city from a given city.

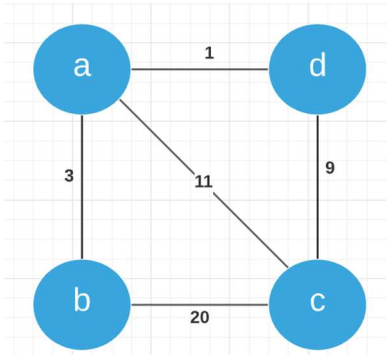
Provide an efficient[1] algorithm. Justify the running time of your algorithm. Show how your algorithm works on a small example of 4 cities.

Start at the center node. Have a graph where every city is another node. When you conquer a city, you choose the node with the lowest rate of cost of maintaining supplies. If a minimum edge increases, then the tree must be updated.

- We are presented with a graph that can be changed and updated at will
- Must find the shortest path to find an adjacent node
    - We can use Prim's algorithm starting from A to determine which shortest path we could take.
    - After conquering a node, we can search the surrounding edges and update their cost relative to the adjacent conquered node we
    - Any vertex marked 0 means the city is conquered and requires no cost
    - If we own node a and node b and want to conquer node c, then we would check MIN{(a, c), (b, c)}
- When an edge updates we only look for one scenario
    - If an edge between two conquered nodes updates, it doesn't matter since we have already conquered the two nodes
    - If an edge between two unconquered nodes updates, it doesn't matter because we will look at that edge once we take one of the two nodes
    - We have to look at the scenario where an edge between a conquered node and an unconquered node updates
- We can look at the paths given to us by conquered nodes as a possibility for reduced edge updates.
    - We would have to update our MST if the edge cost for unconquered nodes changes
    - We do not have to change for paths between two conquered nodes as they would not be necessary.

- Whenever we conquer a node, we will look at all the edges of that node and update our cost adjacency array if the new edge found is smaller than our recorded cost.
- The runtime of the algorithm would be the runtime of running Prim's algorithm alongside the cost of having to search for the minimum adjacent node relative to our conquered nodes
    - The cost of Prim's algorithm is O(|E| + log |V|) but if an edge changes for every iteration, we would have to search the graph for any new lower cost adjacencies to update the binary heap.
    - The cost to search for a potentially lower cost would be O(|E|) time with the cost to update the binary heap to be O(log V) time
    - The run time would eb O(E log V * (|E| + log |V|) time.

If we have 4 nodes where we start at node a:



We would search the adjacencies at node a and get the following list:

COST[N]:

A	B	C	**D**
0	3	11	**1**

We see that D has the lowest cost, so we conquer D and incorporate it. We search D for its adjacencies and incorporate them into the table if it's smaller than A's route.

- D is adjacent to C
    - (d, c) = 9
    - (d, c) < (a, c) == 9 < 11
- D's distance is changed to 0 to indicate that we own node D now.
- C's distance has been reduced to 9 since (c, d) has a smaller cost than (a, c)

Smallest cost representation:

A	**B**	C	D
0	**3**	9	1

We look at the matrix and see that (a, b) has the smallest cost, so we incorporate node B now.

- We change B's cost to 0 to indicate that we own B.
- We look at B's adjacencies for any other paths.
    - B is adjacent to c
    - (d, c) < (b, c) == 9 < 20
    - We do not update the matrix adjacency table because the cost is greater than the one we already recorded.

Cost[V]:

A	B	**C**	D
0	0	**9**	0

- C is the only node remaining, so we incorporate C, ending the algorithm.
- Our path would be (A, D), (A, B), (D, C)

We can use Prim's algorithm to build the initial matrix adjacency list starting from node A.

PRIM(G,w,r)

Q = ∅

Let ADJ[1 … V, 1…V] be a new array

for each u ∈ G.V

ADJ[u, 1 …V] = ∞

ADJ[u, u] = 0

u.key = ∞

u.π = NIL

Q[u] = u.key

DECREASE-KEY(Q,r,0) //r.key = 0

//Start with Algorithmica

MEMOIZED-PRIM(G, ADJ, Q, w, r)

MEMOIZED-PRIM(G, ADJ, Q, w, r)

while Q ≠ ∅

If new weight X(u, v) was detected

If u.key == 0 **and** v.key > 0 //U is conquered

```
        If (X(u, v) < w(u, v))    DECREASE-KEY(u, v, X(u, v))
        Else //Check for any adjacencies that have a smaller cost to a different conquered city
          MIN = X(u, v)
          for each A ∈ G.Adj[u]
            If A.key == 0 and w(a, v) < x(u, v)     //If a is conquered and (a, v) has a lower cost than (u, v)
                MIN = w(a, v)
          v.key = MIN
          V.π = A
          INCREASE-KEY(Q, v, MIN)
    Else If u.key > 0 and v.key == 0        //V is conquered
        If (X(u, v) > w(v, u))
          MIN = X(v, u)
          for each A ∈ G.Adj[u]           //Find the smallest edge in the graph
            If (A.key == 0 and w(a, u) < MIN)
                MIN = w(a, u)
          u.key = MIN
          u.π = A
          INCREASE-KEY(Q, u, MIN)
        Else      DECREASE-KEY(Q, u, X(v, u))

u = EXTRACT-MIN(Q)
u.key = 0
    for each v ∈ G.Adj[u]
        w(u,v) < v.key
        v.π = u
        DECREASE-KEY(Q,v, w(u,v))
```

Brandon Vo

9. (3 bonus points) Think of a good exam question for the topics in this lecture.

When presented with an undirected graph where every city is connected to every other city by a road. Find 3 sets of cities that have the largest section of free space shared by those 3 cities' roads. (Use a graph to find the area between 3 cities).