```
Let min_cost = 0
Let root.cost = 0
Let L be an empty set // stores left nodes for shortest path
Let R be an empty set // stores right nodes for shortest path
Let P be an empty set // represents shortest path
Let P[0] = root
Let L[0] = root
Let R[0] = root
MinimumCost(G=(V, E), P, n, p, q, root, min_cost) {
    if all edges have been visited, then
        return p * L.length()-1 + q * R.length()-1
    else if root is null {
        return
    }
    else {
        if the left edge has not been visited, then
            mark edge as visited
            Let root.left.cost = min_cost + p
            add root.left to L
            add root.left to P
            return MinimumCost(G=(V, E), P, n, p, q, root.left, min_cost +
p)
        if the right edge has not been visited, then
            mark edge as visited
            Let root.right.cost = min_cost + q
            add root.right to R
            add root.right to P
            return MinimumCost(G=(V, E), P, n, p, q, root.right, min_cost
+ q)
        if both leaves are null, then
            if min_cost > root.cost, then
                for vertex in P {
                    remove vertex in L not found in P
                    remove vertex in R not found in P
                    if vertex not in R, then
                        add vertex in P not found in L to L
                    if vertex not in L, then
                        add vertex in P not found in R to R
                }
            Let min_cost = Math.min(min_cost, root.cost)
```

```
    }
}

2) LocalMinimum(f[a....b]) {
    Let m = Math.floor(a+b/2);
    if (f[m-1] > f[m] and f[m+1] > f[m]) {
        return f[m];
    }
    else {
        if (f[m-1] > f[m] and f[m+1] < f[m])
            return LocalMinimum(f[m+1...b]);
        else if (f[m-1] < f[m] and f[m+1] > f[m])
            return LocalMinimum(f[a....m]);
    }
}

3) MaxBandwidth(G=(V, E), s, d) {
    Let S be an empty array // Stores shortest path
    Let V be an empty array // Stores current path
    Let max_bandwidth = 0 // Represents highest bandwidth
    Let s.bandwidth = 0
    Let S[0] = s
    Relax(s)
    while d is not in S {
        Let max = 0
        Let vertex = null
        for vertex v in V {
            if max < v.bandwidth, then
                Let max = v.bandwidth
                Let vertex = v
        }
        remove vertex from V
        add vertex to S
        Relax(vertex)
    }
    return S
}
```

```
Relax(curr) {
    for v adjacent to curr that is not in S {
        Let max_bandwidth = curr.bandwidth + bw(curr, v)
        if v is not in V, then
            Let v.bandwidth = max_bandwidth
            Let curr = max_bandwidth;
            add v to V
        if v is in V and max_bandwidth > v.bandwidth, then
            Let v.bandwidth = max_bandwidth
    }
}
```

Proof:

Theorem: Dijkstra's Algorithm will find the path with the maximum bandwidth from s to d.

Proof:

Consider an arbitrary weighted connected undirected graph $G=(V, E)$, with no negative bandwidths and $s \in V$ and $d \in V$1.

Let S be a list of vertices and bandwidths assigned by Dijkstra's algorithm in the order the vertices
are relaxed by Dijkstra's algorithm. Let O be a list of vertices and their maximum bandwidths in the same order as S.peek

Let i be the index in S and O where a disagreement occurs such that s_i =/= o_i. Since O is optimal, then the only possible outcome is that o(i) >= s(i) or else there is a contradiction in which the algorithm is buggy because O always returns the path with the maximum bandwidth. Let C be the vertices prior to v upon which S and O agree.

Consider any path v other than the one found by S. Any such path must start in C, leave C via another vertex u and then return
to v with 0 or more vertices in between.

Since u is connected to a vertex in C, it must be a vertex for which we have an estimate. Since v is the vertex with

maximum estimate u must have a lesser (or equal) estimate to v. Since the cost to reach u is already less than the cost to reach v,
we can assume that u is not optimal because there is only one path with the maximum bandwidth. As such, if there are an arbitrary number
of vertices in between u and d as there are between v and d, then only v will provide the optimal solution.

4) Refer to Myers.java file for results

5) Theorem: Select elements in A in descending order such that the maximum of a_i * log(b_i) is always selected first given that B is in descending order as well.

Proof: Consider an arbitrary instance of the payoff problem.

Let B[1...n] represent a set where its elements are sorted in descending order.
Let S[1...n] represent the selected set A in descending order that is used in our algorithm to produce the maximum payout.
Let O[1...n] be some optimal set of A in descending order that produces the maximum payout.
Let i be the first instance where S and O disagree on.

If s_i = o_i, there is no disagreement. The algorithm holds.
If s_i < o_i, then there is a contradiction because our algorithm always selects the element in A
with the highest value first provided that set B is in descending order.
If s_i > o_i, then consider all of the elements in O from i+1 to an arbitrary j. Because O selects elements with
the highest value, then all elements in O[i+1....j] < O[i] must be true. As such, the highest product of a_i * log(b_i) will always be chosen which makes this a valid solution.
As such, we can replace o(i) in O with s(i) to create a new solution O'
O' is valid because it will still select the elements with the highest element in A first.
O' is optimal since the highest payout is still produced according to the algorithm.

```
   Now O and S agree at point i. Repeat this argument for every i in
disagreement to show that S = O for some optimal set O.
```