**Question 1** (List Multiplication Runtime, 30 points). Compute the lengths of the shortest paths from $A$ to each other vertex in the following weighted graph:

We note that this graph is a DAG with edges going from early letters alphabetically to later ones, and can thus use the algorithm for computing shortest paths in DAGs.

The shortest path to $A$ is length 0.
The only path to $B$ is through $A$ and has length $-1$.
The only path to $C$ has length 1.
The only path to $D$ is through $B$ and has length $-1 - 1 = -2$.
The only path to $E$ is through $C$ and has length $1 + 3 = 4$.
The paths to $F$ could be through $C$ (best length $1 + 1 = 2$), through $E$ (best length $4 - 2 = 2$) or through $D$ (best length $-2 + 2 = 0$). So the shortest path is length 0.
The path to $G$ must come through $E$ and have length $4 - 1 = 3$.
The path to $H$ could come through $D$ (best length $-2 + 1 = -1$) or $F$ (best length $0 - 2 = -2$). So the shortest path is of length $-2$.
The path to $I$ must come through $G$ and thus the shortest length is $3 + 0 = 3$.
The path to $J$ could come through $H$ (best length $-2 + 2 = 0$) or through $I$ (best length $3 - 3 = 0$). So the shortest path length is 0.
**Question 2** (Increase Finding, 35 points). Suppose that you are given an array $A$ of length $n$ with $A[n] > A[1]$. Give an algorithm to find an index $i$ so that $A[i+1] > A[i]$. For full credit your algorithm should run in time $O(\log(n))$ or better.

We proceed by divide and conquer. The basic idea is to maintain a pair of indices $i < j$ with $A[i] < A[j]$. At each step our algorithm queries an index immediately between $i$ and $j$ and will then be able to reduce the size of the search space by half. The algorithm is as follows:

```pseudo
FindIncrease(A, i, j) \ \ Assume \ i < j \ and \ A[i] < A[j]
  If j = i + 1
    return i
  k <- [(i+j)/2]
  If A[k] > A[i]
    return FindIncrease(A, i, k)
  Else If A[k] < A[j]
    return FindIncrease(A, k, j)
  Else \ This is impossible unless A[i] > A[j]
    return 'error'
```

For correctness, note that if $j = i + 1$ and $A[j] > A[i]$, then $i$ is a valid output. Otherwise, if $A[j] > A[i]$ it cannot be the case that $A[i] \geq A[k] \geq A[j]$, and therefore one of the first two branches of the if statement must be processed. If $A[k] > A[i]$, then the assumptions for the recursive call are satisfied, and so it will return an appropriate index. Similarly, the latter case will also satisfy assumptions. Furthermore, since at each step the length of the interval being searched decreases, the algorithm will eventually terminate. This proves correctness. To run this on the full array, simply call `FindIncrease(A, 1, Len(A))`.

To analyze the runtime, note that a single call to our algorithm does constant overhead, and then makes a single recursive call to an instance with $j - i$ half as large. Thus, if $T(n)$ is the runtime of this algorithm on an instance with $j - i = n$, then

$$T(n) = T(n/2) + O(1).$$

So by the master theorem, the runtime of the algorithm is $O(\log(n))$. 
**Question 3** (Around the World, 35 points). *Around the World Airlines is offering a deal on trips around the world. The rules are that you can take any series of flights, but that each must travel only Westward (in addition to North and South) and the route must end in your starting city having travelled around the world exactly once. Roxy is considering taking them up on this offer. She has a map of all the cities that are covered by airline, and all of the flights between them that would be allowed as part of this deal. To each flight there is an associated cost, but to each city Roxy associates a value to going there. Devise an algorithm to determine if there is some allowable flight plan for Roxy, starting and ending at some city *s*, where the total value of all cities she visits exceeds the total cost of all of the flights she takes.

For full credit, your algorithm should run in time proportional to the total number of cities plus the total number of possible flights.

We will turn this problem into a shortest path problem in a graph. As a first pass, we could create a directed graph with cities as vertices and flights as edges, where flights are only allowed if they travel westward and do not cross the longitude of Roxy’s starting city. To allow the trip to be completed though, we will need to create a duplicate of Roxy’s city to reach at the end of the trip. The important fact about this graph is that it is a DAG (as longitude provides a topological ordering). For each vertex in the graph we will compute \( B(v) \), the best possible value over all paths from the start city to \( v \) of the total value of all cities on the path minus the total cost of all edges along the path. It is not hard to see that

\[
B(v) = \begin{cases} 
0 & \text{if } v \text{ is the starting city} \\
\max_{(u,v) \in E} B(u) - \ell(u,v) + \text{val}(v) & \text{otherwise}
\end{cases}
\]

In particular, because the best value of a path to \( v \) coming immediately from some other vertex \( u \), is the best value of the trip to \( u \) plus the value of \( v \) minus the cost of the last flight. The best we can do is to optimize this over all \( u \).

Once the graph is created, we can compute \( B(v) \) for all vertices in topological order. Since we will already have computed \( B(u) \) for all \( u \) upstream of \( v \) before having computed \( B(v) \), this computation can be done in the natural way. We then merely need to check whether or not \( B(t) \) is positive where \( t \) is the second copy of Roxy’s starting city. The cost of computing \( B(v) \) is \( O(1 + \deg(v)) \). Summing over all vertices, the total runtime is \( O(|V| + |E|) \) as desired.