**Answer 1: (Manna)**

Partitioning the $n$ input numbers into two equal halves is same as finding if the input $n$ numbers have a subset summing $S/2$ (the sum can be calculated in linear time). So, partitioning is a particular case of the general problem of “Subset-sum”, where one needs to find out if there exists a subset with a sum of some $K$.

First, if not sorted already, sort the input $n$ numbers. Put them in $A[n]$.

Now, create a Boolean array $B$ of $n+1$ columns and $K+1$ rows. If $B[i,j] = Y$, that means there exists a subset summing $i$ using numbers only from the first $j$ integers among the $n$ available ones (it does not mean that you will have use all $j$ of them – it just means that integers $A[j+1] \ldots A[n]$ are not required for summing $i$). So, you can see that if $B[i,j] = Y$, so will $B[i,j+1] \ldots B[i,n]$.

Our initial table would look like something like this (the initializations are trivial)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>….</th>
<th>$N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>….</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$K-2$</td>
<td></td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$K-1$</td>
<td></td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$K$</td>
<td></td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now, you fill out the columns one by one (not row by row). So, try to visualize that we are adding the input numbers one by one and watching what new sums we can achieve with this new integer.

For each row entry $B[i,j]$ in that new column (say column $j$), do the following:

**If** ( $(B[i,j-1] = Y)$ or $(B[i-A[j], j-1] = Y)$ ), **then** $B[i,j] = Y$ else $N$.

The first condition has already been explained above. It exploits the fact that if $B[x, j] = Y$ for any number $x$, that means there exists a subset (say $S_1, S_2, \ldots, S_m$ for some $m \leq j$) who add up to $x$. So, the value of the entries must be $Y$ too for all $B[x - S_1], B[x - S_2]$ etc.

The second condition needs attention to understand. Basically, it means that if one can come up with a subset summing $K$ using the first $j$ numbers in $A$ if and only if it is possible to come up with a subset summing $K-A[j]$ using the first $j-1$ integers in $A$.

This way, one can fill out the table in $O(nK)$ time, and if $B[K,n] = Y$, that means there exists a subset summing $K$. 
Some scope for optimizations:

- If the desired sum $K$ is less than some of the input numbers, then discard those big numbers as they are of no use to us for obvious reason.
- While filling out the rows for column $j$, if sum of first $j$ integers is $X$, all row entries below $X$ in that column (i.e., $B[X+1,j] \ldots B[K,j]$) would be $N$ too.

For $j = 1$ to $n$ do
  For $i = 1$ to $K$ do
    If ( $(B[i,j-1] = Y) \text{ or } (B[i-A[j], j-1] = Y)$ )

As you can see, the algorithm is very concise and elegant. It runs in $O(nK)$ time, so the partitioning problem would run in $O(nS)$ time.

Does that mean this is linear?
No.

Is it Quadratic?
No again!

Then, at least is it polynomial in some powers of $n$?
Nope!!

Then what is it?
It’s exponential in the input size.

Why????
You will know when you will understand NP-Completeness.

Then how come it is considered an “efficient” solution?
Well, you were asked to give an algorithm “as efficient as possible”, and this is as good as it gets. If you can find an alternative solution polynomial in input size, you have solved the greatest outstanding problem in computer science, i.e., by proving $P = NP$!! 😊
Answer 2: (Manas)

Given:

Graph $G = (V, E)$
The weight function $w: E \rightarrow R$
Source vertex $s$

Algorithm:

We use the array $d$ to store the shortest path estimates and the array $\pi$ to store the predecessor vertices.

```
BELLMAN-FORD (G, w, s)
1 // Initialize shortest path estimates and predecessor vertices
2 for each vertex $v \in V$
3     $d[v] \leftarrow \infty$
4     $\pi[v] \leftarrow$ NIL
5 $d[s] \leftarrow 0$
6 // Bellman-Ford algorithm (as you already know)
7 for $i \leftarrow 1$ to $|V| - 1$
8     for each edge $(u, v) \in E$
9         // Relaxation
10         if $d[v] > d[u] + w(u, v)$ then
11             $d[v] \leftarrow d[u] + w(u, v)$
12             $\pi[v] \leftarrow u$
13 // Negative cycle detection (extra thing – what we would be looking for)
14 for each edge $(u, v) \in E$
15     if $d[v] > d[u] + w(u, v)$ then
16         return FALSE
17 return TRUE
```

Explanation: Lines 15 – 19 perform the negative cycle detection. The intuition here is that after running the Bellman-Ford algorithm if we can further find a shorter path after considering an edge in the graph then we have a negative cycle.

Time complexity: Easy to see from construction that it is $O(VE)$
Answer 3: (Rajendra)

This problem would have been the same as the Single-Source-Shortest-Path (known as the Bellman-Ford algorithm) had the vertex weights been zero.

So, how to convert (or, reduce) this problem into one that has zero vertex costs?

Use a technique called “node splitting”. Here we explain one of the ways to do this in detail. For every node \( v \) of \( G \), find its degree[\( v \)] (number of edges incident on it, easily calculated from the Adjacency list or matrix). Create a new graph \( G' \) where replace each vertex \( v \) in \( G \) by degree[\( v \)] – number of vertices of zero weight. Connect all these degree[\( v \)] vertices with each other with edges weighing same as the original node weight. So, effectively every vertex \( v \) in \( G \) of node weight \( X \) and degree \( n \) is replaced with a clique of size \( n \) and edge weight \( X \) in \( G' \).

If the vertex \( v \) in \( G \) had degree of \( n \), that means \( n \) edges were incident on it. Now, in \( G' \) we have \( n \) vertices instead of just one in \( G \), so one can attach each of the \( n \) original edges to exactly one of those \( n \) vertices in \( G' \). As an example, suppose \( G \) had a node \( v \) of weight 10 and degree 3. Now, in place of \( v \), \( G' \) would have a 3-clique (basically, a triangle) of edge weight 10 and each zero-weight node in this clique would have the degree of 3.

First, you may sort the adjacency list (this will not change the graph in any way).

For every node \( v \) in \( V \) with nodes \( u_1, u_2, \ldots, u_m \) in its adjacency list \( L_v \), split the node \( v \) into \( m \) nodes by creating nodes \( v_1, v_2, \ldots, v_m \) in \( G' \) with their adjacency lists containing exactly one vertex (\( u_1 \) for \( v_1 \), \( u_2 \) for \( v_2 \), \ldots, \( u_m \) for \( v_m \) etc).

Thus, \( v \rightarrow u_1, u_2, \ldots, u_m \) becomes \( v_1 \rightarrow u_1, v_2 \rightarrow u_2, \ldots, v_m \rightarrow u_m \)

So, now in the modified representation of \( G' \), every adjacency list contains a single node. However, there is a catch: the nodes on the LHS are represented in subscripted format (like \( a_i \)), but the single nodes in the adjacency list are represented in non-subscripted format (like \( b, c, d \)). We resolve this in the following way.

For every node \( v_i \) in \( V' \) (of \( G' \)) do
- For the vertex \( x \) in its adjacency list do
  - Go to adjacency list of \( x \), which would have \( v \) as the only member
  - If it is non-subscripted, make it subscripted \( v_i \)

This way, all the previous edges have been replaced. Now, add the new edges of the clique by adding all \( v_j \)'s to the adjacency list of \( v_i \), where \( i \neq j \).

The whole transformation (a basic sketch of which has been given) runs in \( O(V+E) \) time.
Once the new graph is ready, you can just apply the Bellman-Ford algorithm since no node has any weight here – it will run in $O(VE)$ time. The assumption of absence of negative cycle also applies here.

(A common mistake would be split the node into two nodes – incoming & outgoing. This would work for directed graphs, not for undirected graphs – convince yourself!)