Solution for Islands

We rephrase the problem in terms of graph theory, treating the islands as vertices and bridges as edges. Then the condition of taking a ferry becomes that you cannot add (and immediately traverse) an edge to a vertex that is already connected to the your current vertex.

Consider the connected components of the graph. Since you cannot use ferry to jump within a connected component, your track through the component must form a simple path. And since you can start and end at any vertex of the connected component, the problem reduces to finding the longest weighted path in each connected component. The sum of these values over all connected components gives the desired answer.

This becomes the longest path problem, which is NP-complete for general graphs. It can be done using dynamic programming in $O(2^V E)$ time. To do better, we need to exploit the structure of the graph. As we are dealing with connected components only, we may assume the graph is connected.

It is quite intuitive to see, and not all that difficult to prove the following lemma:
(Several methods exist, with induction being the easiest and ugliest way to go.)
For any graph, any two of the following three statements imply the remaining one:
1. the graph has no cycles
2. the graph is connected
3. the number of edges is 1 less than the number of vertices in the graph.

(And in all three cases, the graph in question is a tree.)

Let the number of vertices in the connected component be N, then it must also have N edges, one associated with each vertex. From the lemma we get that it must contain a cycle. However, if we remove any edge on the cycle, we are not removing any connectivity, as any walk using that edge can go the 'other way' along the cycle. Thus after we remove the edge, we get a connected graph with N vertices and N-1 edges. By the lemma, there are no cycles left in the graph. Therefore the graph has exactly one cycle. Let C be the number of vertices on the cycle.

Note that this observation immediately yields a O(NC) solution for the component: No path can contain all edges of the cycle. Thus for each edge of the cycle, we can try to remove it and calculate the diameter of the resulting tree. The diameter of a tree on N vertices can be calculated in O(N). One tricky way to do it is to start from any vertex A, find the furthest vertex B from it, then find the furthest vertex C from B, and return the distance of B and C. (The proof of correctness of this algorithm is somewhat tricky.)

We will now show how to get a sub-quadratic solution. Assume that we label the vertices on the cycle V_1 to V_C, in order. Then the edges of the cycle are (V_1,V_2),(V_2,V_3)....(V_{C-1},V_C) and finally (V_C,V_1). If we remove these edges, we get a cyclic sequence of rooted trees, one at each of the vertices. (Some of the trees can just be single vertices.)

There are 2 cases for the optimal path: either it lies entirely within one of these trees, or it crosses 2 trees by taking a section of the cycle. We deal with these cases separately.

Case 1: Within the same tree.

This reduces to the problem of finding the longest path in a tree. It can be done in O(N) time total by using the algorithm described earlier on each of the trees.

Case 2: Suppose the two trees involved are the ones rooted at V^i and V_j (where i<j).

Then the path within the trees should be as long as possible. So for each of the cycle vertices, we will compute the length of the longest path from it to some leaf in its tree. We will denote the length of the longest such path from V_k as $maxLen_k$.

There are 2 ways of traveling from V_i to V_j: $(V_i, V_{i+1}, ..., V_j)$ and $(V_i, V_{i-1}, ..., V_1, V_C, V_{C-1}, ..., V_{j+1}, V_j)$. If the edge from V_i to V_{i+1} has length L_i, the cost of these 2 paths are $L_i+...+L_{j-1}$ and $L_j+...+L_C+L_1+L_2+...+L_{i-1}$, respectively.

Note this is almost identical to partial sums. We will track the partial sums of the sequence L_i using SL_i defined as follows: $SL_1=0$, $SL_{i+1}=SL_i+L_i$. Now if we set $S=L_1+....+L_C$, then the two above sums become SL_j-SL_i and $S-(SL_j-SL_i)$, respectively.

If we iterate over all pairs of V_i and V_j, we get a O(C^2) solution for our problem. However, note that we are simply looking for the following value:

$$\max_{i<j} \{ maxLen_i + maxLen_j + \max(SL_j - SL_i, S - SL_j - SL_i) \}$$

Using some algebra, this can be rewritten as follows:

$$\max(\max_{i<j} \{ (maxLen_i - SL_i) + (maxLen_j + SL_j) \}, \quad S + \max_{i<j} \{ (maxLen_i + SL_i) + (maxLen_j - SL_j) \})$$

Consider the first half,

$$\max_{i<j} \{ (maxLen_i - SL_i) + (maxLen_j + SL_j) \}.$$

We can further manipulate this into:

$$\max_j \{ maxLen_j + SL_j + \max_{i \text{ such that } i<j} \{ maxLen_i - SL_i \} \}.$$

So as we loop upwards on V_j, the set of V_is that we need to consider is only increasing. So the value of $\max_{i<j} \{ maxLen_i - SL_i \}$ can be updated in O(1) time as j increases. Hence, this transition can be computed in O(C) time, where C is the length of the cycle. Combining all pieces, we get a O(N) solution, which is expected to receive full points.

It is also possible to do the transition without decomposing the two maxes. This can be done by using a 'sliding window' along the cycle. (Note the ordering condition of the two vertices would no longer hold in this setup.) In this situation, we need to remove vertices from consideration as well. This can be implemented for example in O(N log N) by using a heap, or even in O(N) by implementing a fixed-range range minimum query using a double-ended queue. See also the solution for Pyramids, IOI 2006, for more ideas on a similar problem.