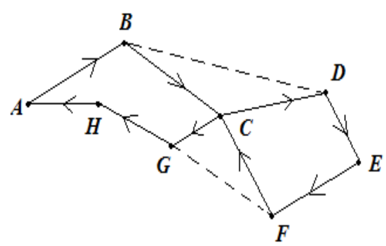
Problem 3 Solution

We claim the following:

The shortest bitonic tour must be a disjoint (can only have common start and end points, but strictly no common points in between) union of exactly 2 paths:

- one strictly going from left to right ($\rightarrow$).
- the other one strictly going from right to left ($\leftarrow$).

Assuming to the contrary, if they are not disjoint, a shorter bitonic tour can always be found by applying triangle inequality, a contradiction, as shown in the following figure.



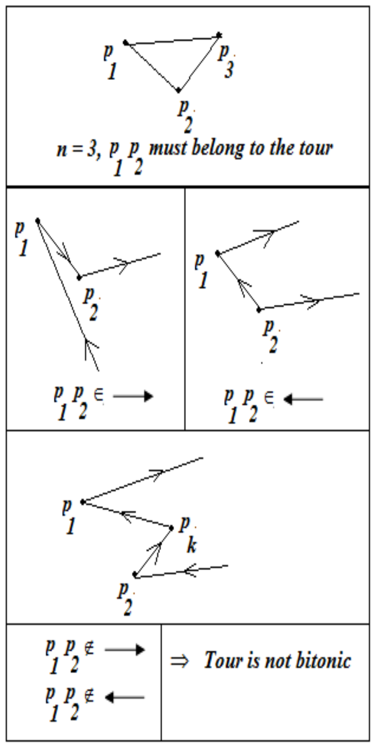
Bitonic tour *(A-B-C-D-E-F-C-G-H-A)*	$\longrightarrow$ *ABCDE* $\longleftarrow$ *EFCGHA* $\rightarrow, \leftarrow$ *non-disjoint*

Triangle Inequality $\Rightarrow FC + CG > FG$ $\Rightarrow$ *length (ABCDEFGHA)* $<$ *length (ABCDEFCGHA)*	
Bitonic tour *(A-B-C-D-E-F-G-H-A)*	$\longrightarrow$ *ABCDE* $\longleftarrow$ *EFGHA* $\rightarrow, \leftarrow$ *disjoint*

Triangle Inequality $\Rightarrow BC + CD > BD$ $\Rightarrow$ *length(ABDEFCGHA)* $<$ *length (ABCDEFCGHA)*	
Bitonic tour *(A-B-D-E-F-C-G-H-A)*	$\longrightarrow$ *ABDE* $\longleftarrow$ *EFCGHA* $\rightarrow, \leftarrow$ *disjoint*

Also we assume that all the points have different x coordinates. First the points are sorted w.r.t. their x-coordinates and let us represent the sorted list of n points by $p_1, p_2, \ldots, p_n$.

We claim that the edge $p_1 p_2$ must be part of any bitonic tour. As seen from the following figure it is obvious that $p_1 p_2$ must belong to the tour when $n = 3$. For general case, it can be seen from the figure that $p_1 p_2$ must belong either to the $\rightarrow$ or the $\leftarrow$ path, otherwise the tour no longer remains bitonic. Being part of a cycle, there must be two disjoint paths from p_1 to p_2. If the edge p_1p_2 is not part of the cycle, then there must be 2 disjoint bitonic tours from p_1 to p_2 for all $n > 3$ in order to complete the cycle, as obvious from the figure. But we want a single bitonic tour, hence the edge p_1p_2 must be part of the bitonic tour. Also, quite obviously, the rightmost point p_n must be the end of the $\rightarrow$ path and beginning of the $\leftarrow$ path.



Now, let's define the following:

$LBP(i, j)$ =Length of the Least bitonic path starting from p_i and ending in p_j, covering all the points $p_i, p_{i+1}, \ldots, p_n$, with $i < j$.

e.g., $LBP(1, 2)$ will represent the length of the least bitonic tour staring at point p_1 and ending at point p_2 covering all the points $p_1, p_2, \ldots, p_n$. We are interested to find $LBP(1, 1)$. Precisely, there will be a $\rightarrow$ path that will start from p_i and end at p_n, where there will be a $\leftarrow$ path that will start from p_n and end at p_j.

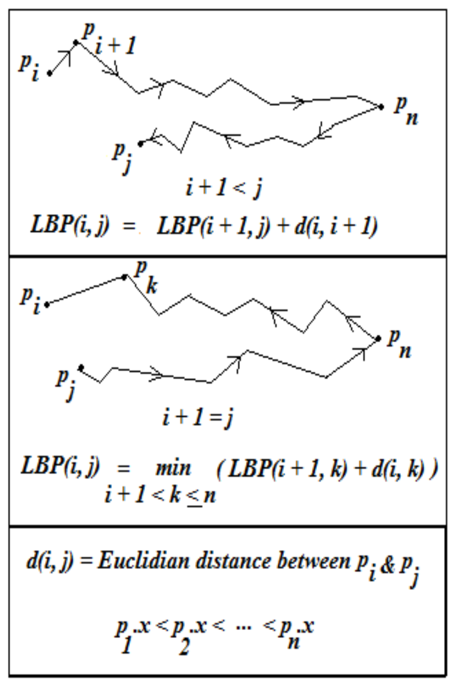
The problem can be recursively formulated as follows:

$$LBP(i,j) = \left\{ \begin{array}{ll} LBP(i+1,j) + d(i,i+1) & \text{if } i+1 < j \\ \underset{i+1<k\leq n}{min} \{LBP(i+1,k) + d(i,k)\} & \text{if } i+1 = j \end{array} \right\}$$

$$LBP(n-1,n) = d(n-1,n).$$

$$LBP(1,1) = LBP(1,2) + d(1,2).$$

Here $d(i,j)$ represents the Euclidian distance between points p_i and p_j.



As can be seen, LBP-LENGTH is $\theta(n^2)$ (including sorting time $\theta(nlogn)$) while PRINT-TSP is $\theta(n)$.

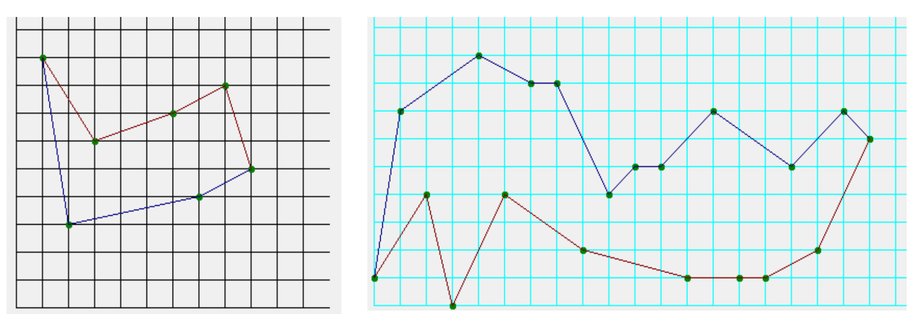


Figure 2: Output of sample implementation for the bitonic TSP

Algorithm 5 LBP-LENGTH: Finds the least bitonic TSP tour length

LBP-LENGTH(P: set of points)

```
Sort the points in P according to their x axis coordinates, with sorted list {p_1, p_2, ..., p_n}.
LBP[n − 1, n] ← d(n − 1, n).
path[n − 1, n] ← n.
for i ← n − 2 to 1 do
    min ← ∞.
    for k ← i + 2 to n do
        if min > LBP[i + 1, k] + d(i, k) then
            min ← LBP[i + 1, k] + d(i, k).
            mink ← k.
        end if
    end for
    LBP[i, i + 1] ← min.
    path[i, i + 1] ← mink.
    for j = i + 2 to n do
        LBP[i, j] ← LBP[i + 1, j] + d(i, i + 1).
        path[i, j] ← i + 1.
    end for
end for
LBP[1, 1] ← LBP[1, 2] + d(1, 2).
path[1, 1] ← 2.
```

Algorithm 6 PRINT-TSP: Finds the least TSP tour

PRINT-TSP(path, i, j, n)

```
if n ≤ 0 then
    return.
end if
if i ≤ j then
    k ← path[i, j].
    print(k).
    PRINT-TSP(path, k, j, n - 1).
else
    k ← path[j, i].
    PRINT-TSP(path, i, k, n - 1).
    print(k).
end if
```

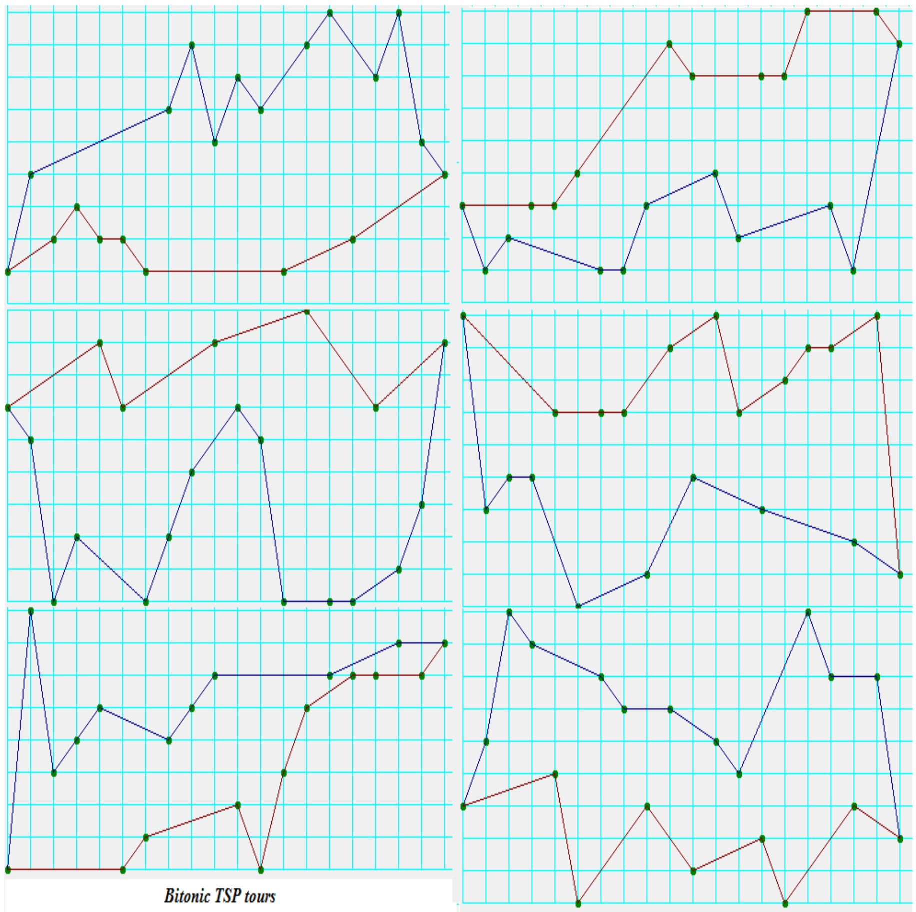


Figure 3: Output of sample implementation for the bitonic TSP