3. (a) We implement the idea of the given fact about having no odd-cycle in a bi-partite graph.

```
BIPARTITE(G)
    pick s ∈ V(G)
    BFS(G,s)
    foreach (u, v) ∈ E(G)
       do if (u.d − v.d) mod 2 = 0
          then return FALSE
    return TRUE
```

This algorithm works since we saw above that if we partition the vertex set according to the distance from a fixed vertex s, then in the case of a graph without odd cycles this will be a bipartition. Since bipartite graphs do not have odd cycles we get the correct answer. If our graph is not bipartite, then no partition will be a bipartition, and so the partition given by BFS also must fail.

The running time of BFS is $O(|E| + |V|) = O(|E|)$, since $|E| \geq |V| - 1$ for connected graphs, and therefore the algorithm BIPARTITE runs in time $\Theta(|E|)$.

Note that depth first search could be used instead of breadth first search in this algorithm.

(b) We can use the CONNECTED-COMPONENTS algorithm (page 441) that gives us the components of G in time $O((|E| + |V|) \log^* |V|)$ and then work with every component separately. However, we can also make a modification to BFS that will allow us to avoid this step. Let $\text{BFS}'(G, s)$ be the algorithm obtained from BFS by omitting the initialization step in lines 1-4. We will only initialize once so that we can keep track of the vertices that we have explored so far. To store the additional information required we will use the extra attributes $u.part$ and $u.cycle$ at each vertex. $u.part$ will be 1 if $u \in V_1$ and 0 otherwise. $u.cycle$ will contain a pointer to the next vertex on the odd cycle to be produced.

```
BIPARTITION(G):
1.  for u ∈ V(G):
2.          u.color == white
3.          u.d = ∞
4.          u.p = NIL
5.          u.part = 0
6.          u.cycle = NIL
7.  for s ∈ V(G):
8.          if s.color == white:
9.              BFS'(G, s)
10.         s.part = (s.d mod 2)
11.         for (u, v) ∈ E(G)
12.             if u.part == v.part
13.                 v.cycle = u
14.                 s = u
15.                 while s.p ≠ NIL
16.                     s.cycle = s.p
17.                     s = s.p
18.                 s = v
19.                 while s.p.cycle == NIL
20.                     s.p.cycle = s
21.                     s = s.p
22.                 s.p.cycle = s
23.                 return FALSE
24. return TRUE
```

If G is bipartite, then every component of G is bipartite. So by the arguments in (a) and (b) the partition found by the second for loop is a good partition. Thus line 12 will always be FALSE and the correct answer TRUE is returned in line 24. Furthermore by accessing *s.part* we can find out for every vertex whether it is in V_1 or V_2 and these set can be determined in time $O(|V|)$.

If G is not bipartite, then the given partition can't be a good partition, so that for some edge line 12 will be true and the correct answer FALSE is returned in line 23. It remains to be seen that an odd cycle is found in lines 13-22. Then v is a pointer to the cycle. Lines 13-22 follow the following argument:

Suppose that G contains no odd cycles. We will show that every component of G (and therefore G itself) is bipartite. Let s be any vertex of G, let $X = \{u \in V(G) : d(u,s) \text{ is even}\}$, and let $Y = \{u \in V(G) : d(u,s) \text{ is odd}\}$. $X \cup Y$ contains all the vertices in the component containing s. We will see that X and Y form a bipartition of this component. Suppose that (u,v) is an edge between vertices in, say, X. Let P_u and P_v be shortest paths from s to u and v respectively. Consider the last vertex on P_v that is also on P_u and call it w. The cycle formed by going from u to w along P_u, then to v along P_v and back to u via (u,v) has length

$$d(u,s) - d(w,s) + d(s,v) - d(s,w) + 1 = 1 + d(s,u) + d(s,v) - 2d(s,w).$$

This length would be odd if u and v were in the same part, but that can't happen since there are no odd cycles.

In line 14-17 the whole path P_u is temporarily put into the cycle. Then in 18-21 we search along P_v until we find a vertex w in P_u. In line 22 w is then reconnected to P_v and in line 13 v is connected to u.

Running time: $O(|V|)$ for lines 1-6. Line 9 is executed exactly once for each component, and has a running time of $O(|V'| + |E'|)$ for the component (V', E'). So lines 7-10 have a running time of $O(|V| + |E|)$. Lines 13-22 have a running time of $O(|V|)$, since every vertex is put into the cycle at most once. Since line 11 is encountered at most $|E|$ times and the lines 13-23 are only executed at most once, the loop 11-23 has a running time of $O(|E| + |V|)$. Hence the algorithm runs in time $O(|V| + |E|)$.

Practice Question 2

- 1. We construct the adjacency list of an undirected graph. Each vertex, labelled by coordinate (x, y), represents a square in the field. There is an edge between two vertices if and only if a player can move from one square to the other in one step. That is, for each vertex (x, y) the four neighbours are $(x-1, y)$, $(x+1, y)$, $(x, y-1)$, $(x, y+1)$; for each neighbor we need to check (1) whether the neighbour's coordinate is out of the bound of the field, which takes $\mathcal{O}(1)$ time, and (2) whether it is an obstacle. In order to efficiently check whether a square is an obstacle, we insert all obstacle coordinates into a hash table, so that checking whether a square is an obstacle would take $\mathcal{O}(1)$ time.

  The runtime of the graph construction procedure involves $\mathcal{O}(mn)$ time for populating the hash table; and for each of the mn vertices, we do $\mathcal{O}(1)$ work at most. Therefore, the total runtime of graph construction is $\mathcal{O}(mn)$.

  2. With the graph constructed, we will do three BFS's, each starting from **H**'s starting vertex, **R1** starting vertex and **R2** starting vertex, respectively. And we compare the shortest path distances from each player to each of the edge (yellow) square. If there there exist a yellow square whose shortest path distance from **H** is smaller than both of those of **R1** and **R2**, then **H** can escape for sure and the algorithm returns `TRUE`. Otherwise, the algorithm returns `FALSE`. The runtime of BFS, as we learned in the lecture, is $\mathcal{O}(|V| + |E|)$. In this problem, $|V|$ is at most mn, $|E|$ is at most $4mn$, therefore the total runtime of the algorithm is $\mathcal{O}(nm)$.