We first analyze Breadth First Search with a rough analysis of the algorithm in order to develop some intuition. We then build on this analysis to provide a more accurate estimate.

**Breadth First Search Rough Analysis**

Here is the pseudocode for the algorithm along with the estimated time complexity for each line:

\[
\text{BFS}(G, s)
\]

1. \textbf{for} \( v \in V \) : \( O(V) \)
2. \textbf{do} \( \text{color}[v] \leftarrow \text{WHITE} \) : \( O(1) \)
3. \( Q \leftarrow \text{MAKE-QUEUE}() \) : \( O(1) \)
4. \( \text{color}[v] \leftarrow \text{GRAY} \) : \( O(1) \)
5. \( \text{ADD}(Q, s) \) : \( O(1) \)
6. \textbf{while} \( Q \neq \emptyset \) : \( O(V) \)
7. \textbf{do} \( v \leftarrow \text{POP}(Q) \) : \( O(1) \)
8. \( \text{color}[v] \leftarrow \text{BLACK} \) : \( O(1) \)
9. \textbf{for} \( u \in \text{Adjacent}[v] \) : \( O(E) \)
10. \textbf{do if} \( \text{color}[u] == \text{WHITE} \) : \( O(1) \)
11. \textbf{then} \( \text{color}[u] \leftarrow \text{GRAY} \) : \( O(1) \)
12. \( \text{ADD}(Q, s) \) : \( O(1) \)

The time complexity estimates in the pseudocode above come from the following observations:

- The first point to consider is the complexity of the operations of the \texttt{Queue} data structure. If we use a \texttt{Linked List} with pointer to the \texttt{tail node} the \texttt{Queue} operations \texttt{MakeQueue}, \texttt{Add}, and \texttt{Pop} can be implemented efficiently in \( O(1) \).

- The other important point is that the body of the \texttt{while} loop, will be executed \( V \) times – the number of vertices in the graph. This may not be clear immediately, but it follows from the fact that each vertex will enter the \texttt{Queue exactly once} and will leave the \texttt{Queue exactly once}. This is ensured by the coloring strategy – once a vertex enters the \texttt{Queue} it is colored \texttt{GRAY} which prevents it from entering the \texttt{Queue} twice. This mean that Line 7, \texttt{Pop(Q)}, which is executed every time through the \texttt{while} loop is executed at most \( V \) times (once per vertex) after which the \texttt{Queue} is empty.

- The final point to note is that the \texttt{for} loop in Line 9 will execute at most \( E \) times. After all, the \texttt{for} loop simply looks at the adjacent edges of \( v \) and at most we may have to examine all edges in the graph.

- Clearly, the initialization step takes \( O(V) \) time since the loop is executed once per vertex and we do constant amount of work per vertex.

Overall the time complexity is

\[
\text{Init(Lines 1 : 2)} + \text{Setup(Lines 3 : 5)} + \text{Search(Lines 6 : 12)}
\]
Individually we have

- **Init**(Lines 1 : 2) takes \(O(V)\) time
- **Setup**(Lines 3 : 5) takes \(O(1)\) time
- **Search**(Lines 6 : 12) can be divided into

\[\text{Finish (Lines 7 : 8)} + \text{Explore (Lines 9 : 12)}\]

That is for each vertex \(v\) we perform a finish step (pop and mark), which takes \(O(1)\) time and we explore its neighbors, which takes at most \(O(E)\) times, i.e. at most all edges need to be explored. Thus per vertex we spend \(O(E)\) time for **Finish** and **Explore**, so for all vertices **Search** (Lines 6 : 12) takes \(O(V \cdot E)\).

Finally, the **BFS** time complexity is

\[\text{Init (Lines 1 : 2)} + \text{Setup (Lines 3 : 5)} + \text{Search (Lines 6 : 12)} = O(V) + O(1) + O(V \cdot E)\]

or \(O(V \cdot E)\) since this is the dominating term.

Our estimate of \(O(V \cdot E)\) suggests that the algorithm is impractical for *dense* graphs. If the graph is fully connected, i.e. every vertex is connected to every other vertex, then we can estimate that \(E \approx V \cdot V\) (actually \(E = V \cdot (V - 1)/2\)), which implies that **BFS** is \(O(V \cdot E) = O(V^3)\), i.e. not practical for large graphs.

**Breadth First Search Precise Analysis**

We now consider a more accurate analysis of **BFS**. The overestimate in our analysis is in **Line 9**. Clearly, we do not need to explore all edges in the graph for each vertex. Instead, for each vertex \(v\) we only explore the adjacent edges for this vertex which is some number \(\text{Adj}_v\).

The more precise analysis breaks **Search** (Lines 6 : 12) by looking at the time spent to process each vertex during its **Finish** and **Explore** steps. Here is the while loop, unwound to show the time spent per vertex:

<table>
<thead>
<tr>
<th>Popped</th>
<th>Finish</th>
<th>Explore</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_1)</td>
<td>(O(1))</td>
<td>(\text{Adj}_{v_1} \cdot O(1))</td>
</tr>
<tr>
<td>(v_2)</td>
<td>(O(1))</td>
<td>(\text{Adj}_{v_2} \cdot O(1))</td>
</tr>
<tr>
<td>(v_3)</td>
<td>(O(1))</td>
<td>(\text{Adj}_{v_3} \cdot O(1))</td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
</tr>
<tr>
<td>(v_V)</td>
<td>(O(1))</td>
<td>(\text{Adj}_{v_V} \cdot O(1))</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>(V \cdot O(1))</td>
<td>(\Sigma\text{Adj}_{v_i} \cdot O(1))</td>
</tr>
</tbody>
</table>

What remains is to see if we can provide an estimate for \(\Sigma\text{Adj}_{v_i}\). We claim that

\[\text{Adj}_{v_1} + \text{Adj}_{v_2} + \text{Adj}_{v_3} + \ldots + \text{Adj}_{v_V} = 2E\]

Even though we do not know the individual terms in the above summation we actually know the overall value of the summation itself. This value is just \(2E\), since every time we look at an adjacent vertex we effectively look at one of the edges \((a, b)\), so each edge \((a, b)\) is looked at twice — once from the point of view of vertex \(a\) and once from the point of view of vertex \(b\).

Finally, we get

\[\text{Search (Lines 6 : 12)} = V \cdot O(1) + \Sigma\text{Adj}_{v_i} = V \cdot O(1) + 2E \cdot O(1) = O(V) + O(E) = O(V + E)\]

This is much better than the first estimate. If the graph is fully connected, in which case \(E \approx V^2\) we get that **BFS** is \(O(V + V^2)\) or \(O(V^2)\), not \(O(V^3)\).
The analysis of Prim’s algorithm is almost identical to the analysis of BFS. The only difference comes from the fact that we use a PriorityQueue, so the complexity of the operations Pop and Add is no longer $O(1)$. Instead

- if we use a LinkedList to implement the PriorityQueue the complexity of Add is $O(1)$ (just add to the end or to the front), but the complexity of Pop is $O(n)$ since we have to traverse the whole list to find the smallest item.

- if we use a BinarySearchTree to implement the PriorityQueue the complexity of both Add and Pop becomes $O(\log n)$

The second option is better so in our analysis we will use $O(\log n)$ for the Queue operations. Effectively, this means that in the algorithm analysis, which is essentially the analysis of BFS, we need to multiply by a factor of $\log n$.

Here is the pseudocode for Prim’s Algorithm:

```
PRIM(G, s)
1    for v \in V
2       do dist[v] \leftarrow \infty
3          parent[v] \leftarrow NIL
4          color[v] \leftarrow GRAY
5      dist[s] \leftarrow 0
6      Q \leftarrow \text{Make-Priority-Queue()}
7    for v \in V
8       do \text{ADD}(Q, s)
9    while Q \neq \emptyset
10       do v \leftarrow \text{POP}(Q)
11          color[v] \leftarrow BLACK
12    for u \in \text{Adjacent}[v]
13       do if color[u] \neq BLACK and dist[u] > \text{weight}(u, v)
14          then dist[u] \leftarrow \text{weight}(u, v)
15          parent[u] \leftarrow v
```

The analysis is as follows:

- **Init**(Lines 1 : 5) takes $O(V)$ — process $V$ vertices with $O(1)$ per vertex

- **Setup**(Lines 6 : 8) takes $O(V \times \log V)$ — add $V$ vertices to the priority queue which costs $O(\log V)$ per vertex

- **Search**(Lines 9 : 15) — we use the same per-vertex analysis as we did in BFS; however, this time **Finish**(Lines 10 : 11) will take $O(\log V)$ because Pop is costlier and similarly **Explore**(Line 12 : 15) will take $O(\log V)$ per adjacent vertex because the priority queue may need to be re-organized.

Here is the table from BFS for the while loop analysis adapted to Prim’s Algorithm to reflect the $\log n$ factor per queue operation:
Finally, we get

\[ \text{Search(Lines 9 : 17)} = V \ast O(\log V) + \sum \text{Adj}_v \ast O(\log V) = \]

\[ V \ast O(\log V) + 2E \ast O(\log V) = (V + 2E) \ast O(\log V) = O((V + E) \ast \log V) \]

Overall, for Prim’s Algorithm we get

\[ \text{Init(Lines 1 : 5)} + \text{Setup(Lines 6 : 8)} + \text{Search(Lines 9 : 17)} = \]

\[ O(V) + O(V \ast \log V) + O((V + E) \ast \log V) \]

which is \( O((V + E) \ast \log V) \), since this is the dominating term.

**Note:** As we discussed, Line 14 represents a relaxation step for vertex \( u \). This means that Line 14 is not \( O(1) \), since the PriorityQueue may need to be re-organized. A simple strategy, if we use a BinarySearchTree, is to delete the vertex and re-insert it which will cost \( O(\log V) \).

**Note:** An implementation of PriorityQueue using a data structure called FibonacciHeap allows for Line 14 to be implemented in \( O(1) \) time (amortized). Will this change the complexity of Prim’s Algorithm?

The same analysis also applies to Dijkstra’s Algorithm, since the two algorithms only differ in what value they compute for the dist[] field. Thus, both algorithms are \( O((V + E) \ast \log V) \).