COMPLEXITY ANALYSIS REVISITED

Analyze the running time of mystery

```
void mystery(int n) {
    for (int i = 0; i < n; i++){
        cout << i; // statement A
        for (int j = i; j < n; j++){
            cout << j; // statement B
        }
    }
}
```

i	time @ i is run $(n-i)$
0	n
1	$(n-1)$
2	$(n-2)$
$n-1$	1

$(n-i)$

$T(n) = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)$

Which statement dominates the running time of mystery?

cout << j will dominate it just runs more often than cout << i;

General Insight

```
void mystery(int n) {
    for (int i = 0; i < n; i++){
        cout << i; // A: Outer loop operation
        for (int j = i; j < n; j++){
            cout << j; // B: Inner loop operation
        }
    }
}
```

When two operations have the same complexity, the one that runs more times dominates.

So total cost = count total executions of the dominant operation.

BFS: Running Time Complexity

$G = (V, E)$ undirected

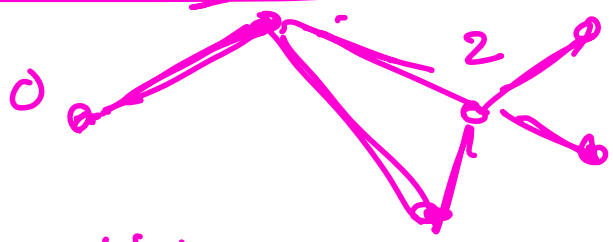
(n vertices, m edges)

Algo exploreBFS (Graph G, vertex s):

- Mark all the vertices as "not visited" O(n) — init
- Mark ***s*** as visited O(1)
- push ***s*** into a queue O(1)
- while the queue is not empty:
  - A: pop the vertex ***u*** from the front of the queue O(1)
    - for each of ***u***'s neighbor (***v***)
      - B: If ***v*** has not yet been visited: O(1)
        - C: Mark ***v*** as visited
        - C: Push ***v*** in the queue O(1)

Parameter: (n, m)

Which statement dominates the running time of exploreBFS?



Insight:

Every vertex is pushed into the queue exactly once (only if it is not visited)

→ every vertex is popped once.

while loop will run n times. because one pop per iteration of the while loop

$$T(A) = T(C) < T(B)$$

$\rightarrow$ B dominates running time

$$T(\text{while loop}) = \text{Number of executions of code B}$$

$$= T(B)$$

Total cost of while loop = Total executions of neighbor check

$= \sum_{u \in V} deg(u) = 2m$

T(B)

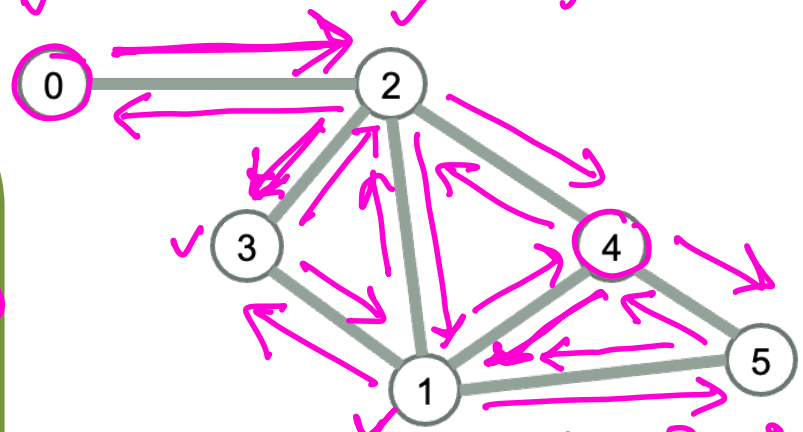
deg(u) = no. of its neighbors

Algo exploreBFS (Graph G, vertex s):

- Mark all the vertices as "not visited"
- Mark ***s*** as visited
- push ***s*** into a queue
- while the queue is not empty:
  - pop the vertex ***u*** from the front of the queue
  - for each of ***u***'s neighbor (***v***)
    - If ***v*** has not yet been visited:
      - Mark ***v*** as visited
      - Push ***v*** in the queue

T(init)

B



$= 1 + 4 + 2 + 4 + 3 + 2$

$= 2 \cdot 8 = 16$

Insight (2): Every edge (u,v) is counted twice

- once when u is popped out of the queue and we v
- once when v is popped out of the que

$T(B) = 2m = O(m)$

$T(init) = O(n)$

$T(overall) = T(init) + T(B)$

$= O(m+n)$

BFS: Space Complexity

What is the Big -O auxiliary space complexity of exploreBFS?

A. O(n)
B. O(m)
C. O(n + m)
D. O(n^2)
E. None of the above

n: number of vertices
m: number of edges

Additional space was used
(1) Vector of visited nodes $\leq n$ $O(n)$ +
(2) Queue. $\leq n$ $O(n)$
$S(n) = O(n)$

- Auxiliary Space complexity: Additional space usage (not including input and output)

exploreDFS(G, s): Time Complexity

```
exploreDFS(v, visited)
    visited[v] = true
    For each edge (v,w):
        If not w.visited
            exploreDFS(w, visited)
```

A

B

C

O(n)

source

Insight: Only call exploreDFS if a vertex has not been visited

→ Visit every vertex exactly once

→ exploreDFS is called once per vertex

Total no. of time exploreDFS is called = n

B dominates

$T(B) = O(m)$

```
exploreDFS(Graph G, vertex s):
    Mark all vertices as "not visited"
    exploreDFS(s, visited)
```

$T(init) = O(n)$

$T(n) = T(init) + T(B) = O(m+n)$

exploreDFS: Space Complexity

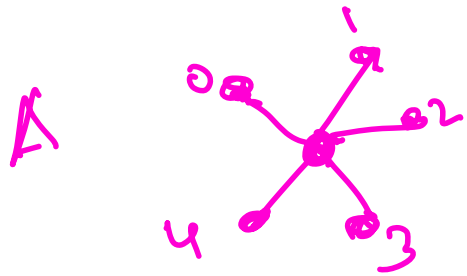
```
exploreDFS(v, visited)
  [visited[v] = true ]
   For each edge (v,w):
    [If not w.visited ]
        exploreDFS(w)

exploreDFS(Graph G, vertex s):
      Mark all vertices as "not visited"
      exploreDFS(s, visited)
```

$S(n) = O(n) + O(\text{max depth of recursion})$ — worst case

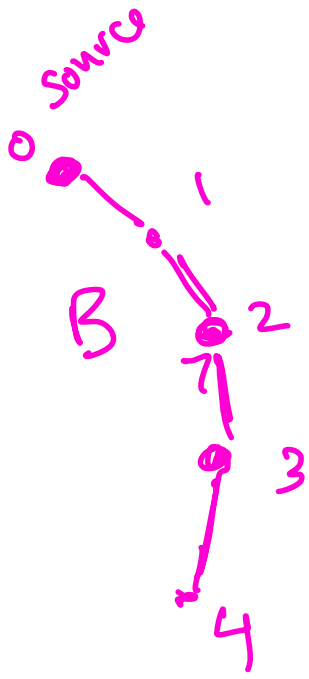
Visited vector

$= O(n-1)$

$= O(n)$



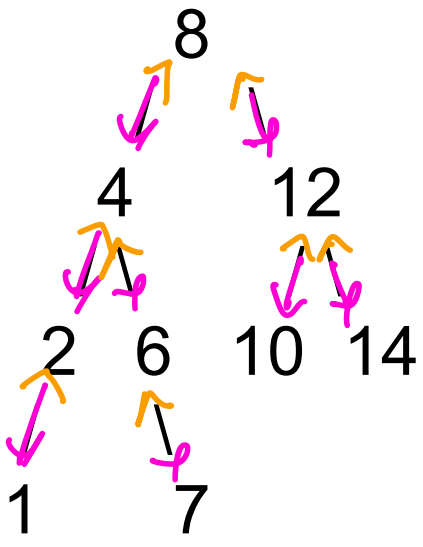
$O(n)$



$S(n) = O(n)$

What is the output of this code for the given BST?

```
void printSetValues(const std::set<int>& s){
    for (int value : s) {
        std::cout << value << " ";
    }
}
```



sets

O(H) = O(logn)

for (auto it = s.begin(); it != s.end(); it++)

cout << *it ; O(1)

1, 2, 4, 6, 7, 8, 10, 12, 14

worst case O(logn)

Call successor

What is the running time complexity for a set with n keys?

```
void printSetValues(const std::set<int>& s){
    for (int value : s) {
        std::cout << value << " ";
    }
}
```

A. O(1) B.O(log n) C. O(n) D. O(nlogn)

Every key (node) is printed exactly once

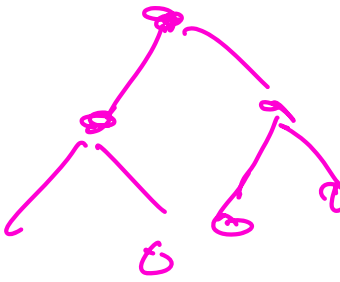
Every edge (u, v) is traversed twice.

① once to traversing into the key v from u.
once when leaving v

② With n keys, we have at most (n-1) edges

Here is why:

Every node except the root has an edge to its parent.



There are n nodes, so n-1 edges total.

Every edge is traversed twice

$$T(n) = O(2(n-1))$$

$$= O(n)$$