PRIORITY QUEUES REVISITED

COMPLEXITY ANALYSIS OF OF GRAPH SEARCH

Tips for studying for the final exam

Detailed tips here: https://ucsb-cs24.github.io/s25/lectures/no-lecture-e02/

- **Do Leetcode sets in reverse (lp05 → lp01)**
  Focus on solving efficiently (~20 min/problem), skip & revisit harder ones.
- **Review lecture slides & handouts after practice**
  Resolve class problems yourself, then compare with annotated solutions.
- **Use recorded lectures for deeper understanding**
  Focus on *why* algorithms work, key patterns, and common pitfalls.
- **Revisit labs & projects for real-world context & usage of C++ STL ADTs**
  Recall what you built, which data structures you used, and why.
- **Make a quick-reference sheet + simulate the exam**
  Track key concepts, then do timed practice—explain your thinking out loud.

C++ Priority Queue ≡ Airport Priority Boarding

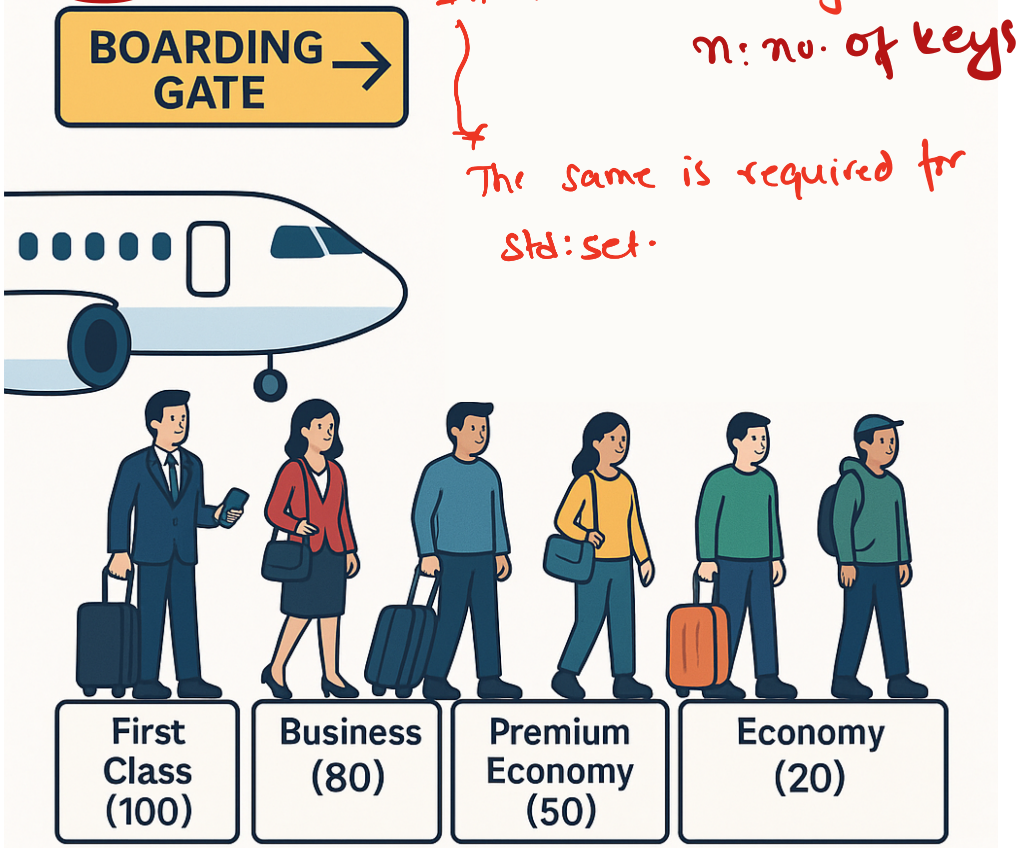
keys

True/False: PQ can only store data for which an ordering can be defined.

In other words keys in a PQ must be comparable in order to prioritize them.

n: no. of keys

The same is required for std::set.




```
priority_queue<int> pq;
// New passengers arrivals
pq.push(20);   ] O(log n)
pq.push(20);
pq.push(80);
pq.push(50);
pq.push(100);

// Whose boarding next?
cout << pq.top();   O(1)

// Next passenger to board
pq.pop();   O(log n)
```

Leetcode practice (LP04)

LP04 (PQ + Hashtables): https://ucsb-cs24.github.io/s25/lp/lp04/

Priority Queues must know problems:

1. Kth Largest Element in an Array (medium): https://leetcode.com/problems/kth-largest-element-in-an-array/description/

2. Top K Frequent Elements (medium): https://leetcode.com/problems/top-k-frequent-elements/description/

* Practice configuring a PQ in different ways using a comparison class

Configuring std::priority_queue

```
template <
      class T,
      class Container= vector<T>,
      class Compare = less <T>
      > class priority_queue;
```

The template for priority_queue takes 3 arguments:

1. Type elements contained in the queue.
2. Container class used as the internal store for the priority_queue, the default is **vector<T>**
3. Class that provides priority comparisons, the default is **less**

Configuring std::priority_queue

```
//Template parameters for a max-heap
priority_queue<int, vector<int>, std::less<int>> pq;

//Template parameters for a min-heap
priority_queue<int, vector<int>, std::greater<int>> pq;
```

Trace the output of this code

```
int arr[]={10, 2, 80};
priority_queue<int*> pq;
for(int i=0; i < 3; i++)
     pq.push(arr+i);

while(!pq.empty()){
     cout<<*pq.top()<<endl;
    pq.pop();
}
```

How can we change the way pq prioritizes pointers?

80

2

10

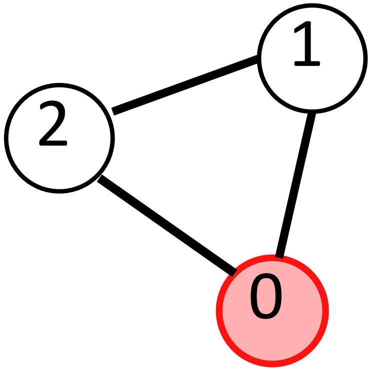
Write a comparison class to get the desired output

```
class cmpPtr{
        bool operator()(int* a, int* b) const {
              return  *a < *b ;
        }
};
int arr[]={10, 2, 80};
priority_queue<int*, vector<int*>, cmpPtr> > pq;
for(int i=0; i < 3; i++)
     pq.push(arr+i);

while(!pq.empty()){
     cout<<*pq.top()<<endl;
    pq.pop();
}
```

```
Output: 80
        10
         2
```

BFS: Running Time Complexity



$G = (V, E)$

Algo exploreBFS (Graph G, vertex s):

- Mark all the vertices as "not visited" $O(n)$
- Mark ***s*** as visited $O(1)$
- push ***s*** into a queue $O(1)$

initialization $= O(n)$

- while the queue is not empty: → while loop runs at most n times. why?
  - pop the vertex ***u*** from the front of the queue
  - for each of ***u***'s neighbor (***v***) Observe that the for loop runs variable times for every iteration of the while loop
    - If ***v*** has not yet been visited: $O(1)$ per iteration
      - Mark ***v*** as visited $O(1)$
      - Push ***v*** in the queue $O(1)$

The count for neighbor visit checks dominates the run time of the code for the inner loop. So just need to count that line

n: number of vertices
m: number of edges

How many times does the while loop run?

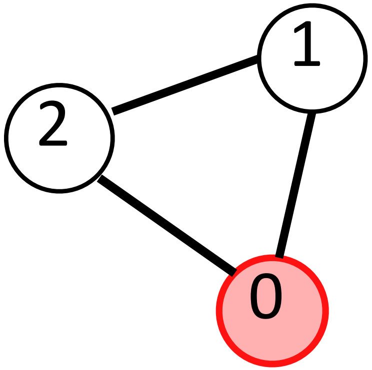
A. n
B. m
C. n + m
D. nm
E. None of the above

Sloppy analysis will bound the for loop iterations by the max no. of neighbors for any node $= m$

To get $O(n.m)$

Pessimistic!

BFS: Running Time Complexity



Algo exploreBFS (Graph G, vertex s):

For each iteration of the while loop, the for loop runs a variable number of times. How should we proceed to analyze the Big-O running time?

- 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

A. Bound the maximum number of times the for loop runs **per iteration** of the while loop

B. Compute the total number of times the for loop runs over **the entire run of exploreBFS**

C. Cannot compute Big-O because running time depends on two parameters (n, m)

To get the complexity just need to count these two lines over the entire run of the algo.

Key questions: ① How many total pop() operations from the queue? ② How many total number of neighbor visit checks?

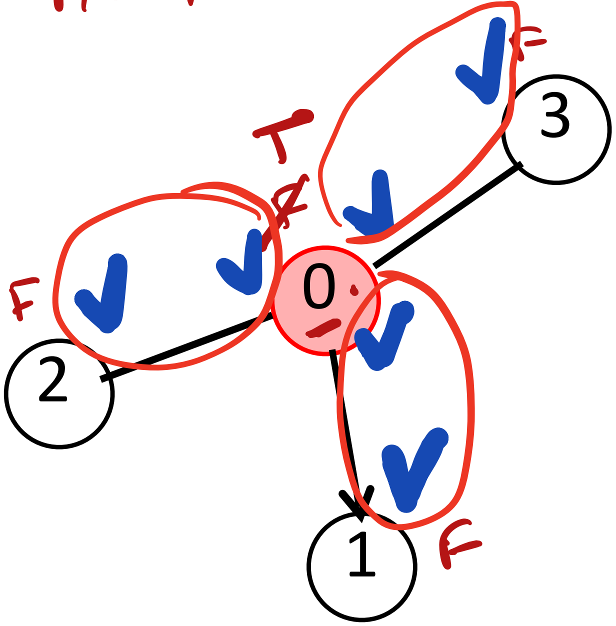
BFS: Running Time Complexity

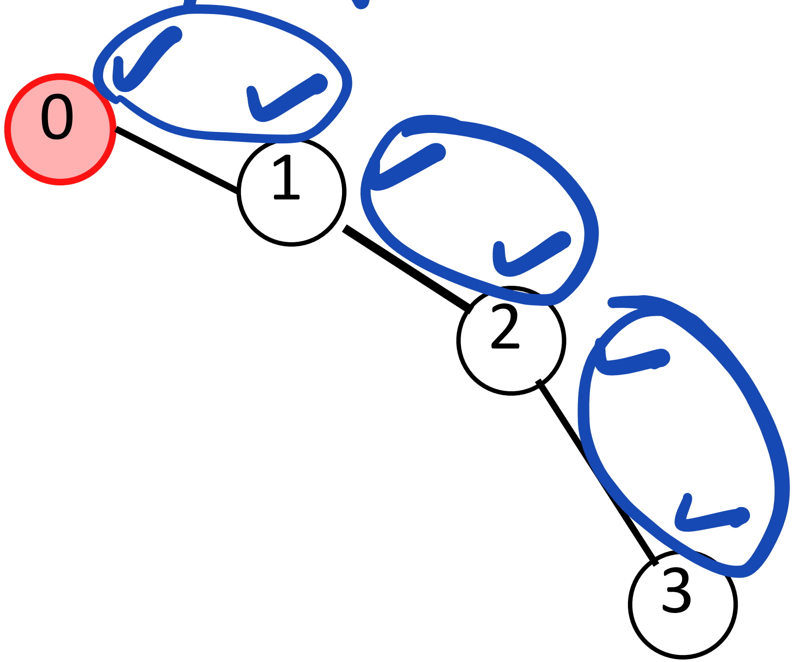
Total number of times the for loop runs over **the entire run of exploreBFS**

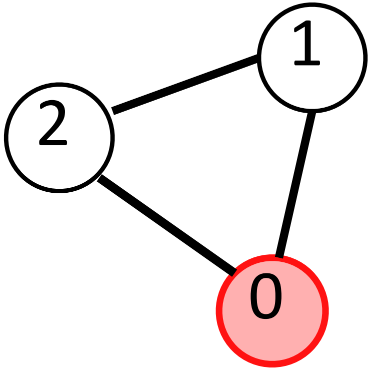
Total number of times each neighbor (u) is checked over **the entire run of exploreBFS**

$n = 4$ $m = 3$

Every edge is checked twice







BFS: Time Complexity

initial visited vector

push source

Initialization : O(n) + O(1)

Overall, the time to do n pops is O(n)

the time to check whether a neighbor is visited is O(m)

Why? Because every edge in an undirected graph is checked twice.

The approach of counting the overall times an operation is performed has showed up many times in past analysis

n: number of vertices
m: number of edges

What is the time complexity of exploreBFS?

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

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

initialize

pop

visit check

BFS Traverse: Space Complexity

visited vector : O(n)

Queue : O(n)

n: number of vertices
m: number of edges

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

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

exploreDFS: Time Complexity

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

O(n) → because each node is visited once, so there are no more than n recursive calls (one per node visited)

O(m)

O(m) : Same reasons as before

Undirected graph: Neighbour visit is done twice per edge
There are m edges, so 2m total = O(m)

Directed graph
Neighbour visit check is done once per edge = O(m)

n: number of vertices
m: number of edges

What is the time complexity of exploreDFS?

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

exploreDFS: Space Complexity

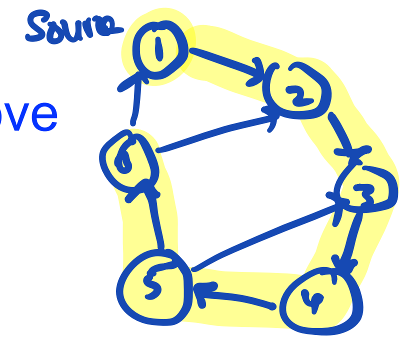
```
exploreDFS(v, visited)
  visited[v] = true
  For each edge (v,w):
    If not w.visited
      exploreDFS(w)
```

n: number of vertices
m: number of edges

What is the worst-case space complexity of exploreDFS?

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

In the worst case, the longest path from source is a chain



Max depth of recursion = Longest path from source

Leetcode practice (LP05)

Max number of fish (medium)
https://leetcode.com/problems/maximum-number-of-fish-in-a-grid/description/

visited 9 3

0	2	1	0
4	0	0	3
1	0	0	4
0	3	2	0

7 5k 5

```
grid =
[[0,2,1,0],[4,0,0,3],[1,0,0,4],[0,3,2,0]]
```

Output: 7

Explanation: The fisher can start at cell (1,3) and collect 3 fish, then move to cell (2,3) and collect 4 fish.

Return *the **maximum** number of fish the fisher can catch if he chooses his starting cell optimally, or* 0 if no water cell exists.

Discuss how you would approach this problem?

Leetcode practice (LP05)

LP05 (BFS/DFS/Divide& Conquer): https://ucsb-cs24.github.io/s25/lp/lp05/

Must know: 1 - 5

1. Find if path exists (easy) https://leetcode.com/problems/find-if-path-exists-in-graph/description/

2. Keys and Rooms (medium) https://leetcode.com/problems/keys-and-rooms/description/

3. Rotting Oranges (medium) https://leetcode.com/problems/rotting-oranges/description/

4. Max number of fish (medium)
https://leetcode.com/problems/maximum-number-of-fish-in-a-grid/description/

5. LCA in a binary tree (medium)
https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/

Extra challenge, can skip or leave for later
6. Minimum Operations to convert number (medium)
https://leetcode.com/problems/minimum-operations-to-convert-number/description/