1. Using i++ in j loop.
2. If and while especially in recusions.
3. Trees/heaps etc: always check for 0 child, 1 child and 2 child.
4. Initialise string: string s(n,’ ‘);
5. Base case: If(Idx==n) use instead **if(idx>=n)**
6. **Incorrect arguments while function call.**
7. In counting scenarios where we maintain cnt and maxi. If all condition true, maxi is never updated to max cnt. Thus at end need to return max(maxi,cnt).
8. Type casting:

long long temp = **2LL** \* min(a, b) + max(a, b);

1. I am taking just greater or lesser. Some questions need : greater than and equal to or lesser than equal to.
2. Queue not initialized with root.
3. Inverted hash mapping.
4. Clear stack:         st=stack<int>();

HashMap

1. Searching value in key value pair:
   1. Maintain a separate hasmap[value]=key.
   2. 2 things:
      1. Hashmap[value]=key -> store count as key
      2. Hash[value]=store set of elements from the map.
         1. Access: \*mpp[value].begin();

Priority Queue:

1. Custom comparator:
2. struct Compare{
3. bool operator()(Node \*a, Node \*b){
4. return a->data > b->data;
5. }
6. };
7. Node\* flatten(Node \*head){
8. priority\_queue<Node \*, vector<Node\*>, Compare> pq;
9. pq.push(head);

Strings:

1. Append – doesn’t add extra memory -> always use this
   1. s.append(“Hello”);
   2. Complexity: if local buffer full -> O(N); repeated appends -> O(1)
2. Substr
   1. S.substr(position,length);
3. Use **string s=” “** and not **string s= ‘ ‘**
4. Sort strings[array]:
   1. Comparator:
5. bool caseInsensitiveCompare(const string& a, const string& b) {
6. string aLower = a, bLower = b;
7. transform(aLower.begin(), aLower.end(), aLower.begin(), ::tolower);
8. transform(bLower.begin(), bLower.end(), bLower.begin(), ::tolower);
9. return aLower < bLower;
10. }
11. sort(arr, arr + n, caseInsensitiveCompare);
    1. sort(Arr,arr+n)
    2. sort(arr, arr + n, greater<string>())
    3. Typecast no into string: to\_string(10);
    4. Insert into vector of strings:
       1. queue<vector<string>> q;
       2. q.push({Word}); // Using list initialization

Sets:

1. Reverse order:
   1. for (auto it = mySet.rbegin(); it != mySet.rend(); ++it) { cout << \*it << " "; }

Comparators:

1. When implementing a comparator in C++, you have **three** main options:
   1. **Function Comparator** (bool Compare(T a, T b))
   2. **Functor (Function Object)** (struct with operator())
   3. **Lambda Function** (auto compare = [](T a, T b) { ... };)
2. **When to Use a Function Comparator?**
   1. **Only use it when passing a function pointer is explicitly required.**
   2. Functions **cannot** be used in STL **containers like priority\_queue, set, or map**, because they don't support function pointers.
   3. Works in sort.
   4. Eg:
      1. bool compare(int a, int b) { return a < b; }
      2. int main() {
         1. vector<int> arr = {4, 2, 7, 1};
         2. sort(arr.begin(), arr.end(), compare);

}

1. **When to Use a Functor (Function Object)?**
   1. ✅ **Use a functor when working with STL containers like priority\_queue, set, or map**
   2. Best for complex comparisons
   3. Eg:
      1. struct Compare {
         1. bool operator()(int a, int b)
            1. { return a > b; // Min-heap (smallest element first) }

};

int main() {

priority\_queue<int, vector<int>, Compare> pq;

}

1. **Lambda comparators:**
   1. For Quick comparisons -> very fast
   2. int main() {
      1. auto compare = [](int a, int b) { return a > b; };
      2. priority\_queue<int, vector<int>, decltype(compare)> pq(compare);

}

Linked List:

1. Passing nodes by reference: void reverse(Node \*head, Node \*&revHead);
2. Many time changing links of nodes is not needed. Just swap the values.

Graphs:

1. Adj Matrix -> list:
   1. For(i=0;i<n;i++){
      1. For(j=0;i<n;i++){
         1. If(adjM[i][j] == 1 && i!=j){
            1. AdjL[i].push\_back(j);
            2. Adj[j].push\_back(i);

}