Coding Interview Prep

Table of Contents

hellooo, I made this as a cheat sheet for myself to look through before interviews & didn't expect to share it then, so a few disclaimers:

1. you might not need to know everything on here, and there might be some missing stuff? (lmk if I should add anything!)
2. the syntax here is just biased towards the language I use & the stuff I was trying to remember. feel free to make your own copies of this doc & replace it with your own, delete what's not relevant, etc
3. there's lots of borrowed content here! I was doing this for myself so I didn't keep track of where everything was from. But generally credits to Interview Cake (1000% recommended) & ofc CTCI

good luck!!! also:
https://www.interviewcake.com/impostor-syndrome-in-programming-interviews

----

During the Interview

Reminders:

- Speak through your process. Show the interviewer that you're thinking, and not just stuck.

Process:

1. Ask questions. Make sure you understand the problem.
2. Finding a solution:

- Does this resemble a problem you've seen before?
- **Do it yourself** (if you were to do what the algorithm's supposed to do yourself, how would you go about it? our brains are smarter than we give them credit for! -- this is also just a starting place)
- Think of the **brute-force. Say it. Why is it bad?**
    - (often complex/inefficient - mention big O **time** and **space**)
- Are there **steps that can be eliminated?**
- Refer to the **algorithmic list below.** Do any of these work?
- Data structure search: Does this fit into a **map/set, stack, queue, vector, graph, tree or linked list?**
    - Graphs are good with connections
    - Maps/sets provide fast lookup + storing frequency of occurrence
    - Remember the properties (FIFO/FILO) of a stack and queue

3. Coding the algorithm

- Start with pseudo (clarify). Change as you go.
- COMMENT YOUR CODE.
- Run a test through the algorithm. Does it work?
- Think of edge cases:
  - Invalid input
  - Different sizes of input
  - Data structure is full
- Talk about complexity
- If steps can be reduced, eliminate repeated steps

Algorithms

Approaches

Greedy algorithm
If steps of brute-force can be eliminated/you can keep track of the value needed as you go

Memoization
If there will be repeat calculations, store each step in a memo (an unordered_map)

Do it Yourself
How would you solve this yourself? Retrace your steps.

Bottom-up Algorithm
When solution is comprised of solutions to the same problem but in smaller sizes

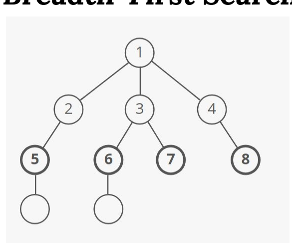
Base-case and build
How would you do this for n=1? What extra step do you take for n=2?

Search

BFS/DFS (graphs, trees)

c++
When a vector is sorted or *almost sorted.*

Breadth-First Search (BFS)



- First, explore all nodes 1 step away. Then, 2-steps away, etc.
- Uses a queue (first in, first out)
- **Advantage:** will find the shortest path between start point and reachable node
- **Disadvantage:** generally requires more memory than DFS

concept:

```
void search(Node root) {
Queue queue = new Queue();
root.marked = true;
queue.enqueue(root); //add to the end of queue

while (!queue.isEmpty()) {
Node r = queue.dequeue(); //remove from front of queue
visit(r);
for each (Node n in r.adjacent) {
        if (n.marked == false) {
        n.marked = true;
        queue.enqueue(n);
        }

}
}


}
```

BFS Syntax (for graphs)

```
// Program to print BFS traversal from a given
// source vertex. BFS(int s) traverses vertices
// reachable from s.
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing adjacency
    // lists
    list<int> *adj;
public:
    Graph(int V);  // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints BFS traversal from a given source s
    void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}
```

```
void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v’s list.
}

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "
         << "(starting from vertex 2) \n";
    g.BFS(2);

    return 0;
```

```
}
```

Depth-First Search (DFS)

- Uses a stack (first in, last out)
- Go as deep as possible down one path, then back up and try a diff. One
- **Advantages:** generally requires less memory than BFS, easily implemented with recursion
- **Disadvantage:** not the shortest path (unlike BFS)

Algorithm

Syntax

For graphs

```
// C++ program to print DFS traversal from
// a given vertex in a  given graph
#include<iostream>
#include<list>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing
    // adjacency lists
    list<int> *adj;

    // A recursive function used by DFS
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V);   // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v’s list.
}
```

```
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to print DFS traversal
    DFSUtil(v, visited);
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"
            " (starting from vertex 2) \n";
    g.DFS(2);

    return 0;
}
```

Sorting

Counting Sort

- If you know the range of the numbers, make a vector/set etc of an equal size and place each element in its corresponding index
- Time efficient, somewhat space efficient
- Exploits O(1) time insertion and lookups in vector
- **Time complexity: O(n)**

Merge Sort

- Recursively split the data in two, sort, and then merge
- **Time complexity: O(nlogn)**

Insertion Sort

- At each iteration, pick out the next element, find the location where it belongs on the sorted list, and insert it there
- **Time complexity: O(n^2) w.c, O(n) best case**

Quick Sort - implementation in C++

- Randomly choose a number to be the pivot, put all the numbers less than the pivot before, and all numbers greater than after
- Recursively repeat this procedure with each of the two piles you made
- **Time complexity: O(nlogn)**

Heap Sort

Cycle:

- Insert data into a heap
- Switch first and last element
- Remove last element from heap
- Heapify

- **Time complexity: O(nlogn)**

Bubble Sort

- Keep passing through array and swapping /sorting every two numbers until no more passes are needed
- **Time complexity: O(n^2) (*inefficient*)**

Radix Sort

Data Structures

Linked Lists

Worst case complexity

- Space: O(n)
- Prepend: O(1)
- Append: O(1)
- Lookup: O(n)

Strengths

- **Fast operations on the ends**. Adding elements at either end of a linked list is O(1). Removing the first element is also O(1).
- **Flexible size**. There's no need to specify how many elements you're going to store ahead of time. You can keep adding elements as long as there's enough space on the machine.

Weaknesses

- **Costly lookups**. To access or edit an item in a linked list, you have to take O(i)O(i) time to walk from the head of the list to the $i$ith item.
- Not cache friendly (as opposed to arrays)

Uses

- **Stacks and queues** only need fast operations on the ends, so linked lists are ideal.

Doubly Linked Lists

- Each node has a next and previous pointer → allows traversing in both directions

Hash tables/hash maps

Strengths: fast lookup, flexible keys
Weaknesses: slow worst-case lookups (collisions), table unordered

Complexity (table)

	avg	worst-case
space	O(n)	O(n)
insert	O(1)	O(n)
lookup	O(1)	O(n)
delete	O(1)	O(n)

Stacks & queues

Queues

- First in, first out
- Implemented with linked lists
  - Enqueue: insert at tail of LL
  - Dequeue: remove at head of LL

Complexity:

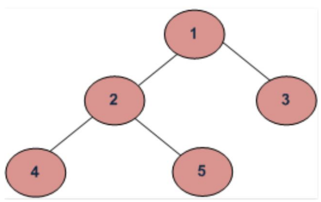
	avg
space	O(n)

enqueue	O(1)
dequeue	O(1)
peek	O(1)

Stacks
- First in, last out
- Implemented with linked list or dynamic array

Trees

Traversals



a) Inorder (Left, Root, Right) : 4 2 5 1 3
b) Preorder (Root, Left, Right) : 1 2 4 5 3
c) Postorder (Left, Right, Root) : 4 5 2 3 1

Binary Trees
- Each node must have 1 or two children.
- **complete binary tree:** every level of the tree is fully filled, except for possibly the last level (which has to be filled left to right)
- **full binary tree:** every node has either zero or two children
- **perfect binary tree**: both complete and fullblue

BSTs
- Already sorted binary tree
- A node is always greater than its left sub-tree, and less than its right sub-tree.

Heap

A Heap is a binary tree T with two properties:
- **relational** property: how keys are stored in T
- **structural** property: what the nodes of T are made of.

Min heap: elements are all smaller than their children
Max heap: elements are all bigger than their children

Insert the elements top to bottom, left to right.

Logarithmic insertion and constant-time access

Tries

```
       root
    /   \    \
    t   a     b
    |   |     |
    h   n     y
    |   |  \  |
    e   s  y  e
 /  |   |
 i  r   w
 |  |   |
 r  e   e
        |
        r
```

```
//C++ implementation of search and insert
// operations on Trie
#include <bits/stdc++.h>
using namespace std;

const int ALPHABET_SIZE = 26;

// trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];

    // isEndOfWord is true if the node represents
    // end of a word
    bool isEndOfWord;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode *getNode(void)
{
    struct TrieNode *pNode =  new TrieNode;

    pNode->isEndOfWord = false;

    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;

    return pNode;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just
// marks leaf node
void insert(struct TrieNode *root, string key)
{
    struct TrieNode *pCrawl = root;

    for (int i = 0; i < key.length(); i++)
    {
```

```
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->isEndOfWord = true;
}

// Returns true if key presents in trie, else
// false
bool search(struct TrieNode *root, string key)
{
    struct TrieNode *pCrawl = root;

    for (int i = 0; i < key.length(); i++)
    {
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            return false;

        pCrawl = pCrawl->children[index];
    }

    return (pCrawl != NULL && pCrawl->isEndOfWord);
}

// Driver
int main()
{
    // Input keys (use only 'a' through 'z'
    // and lower case)
    string keys[] = {"the", "a", "there",
                    "answer", "any", "by",
                     "bye", "their" };
    int n = sizeof(keys)/sizeof(keys[0]);

    struct TrieNode *root = getNode();

    // Construct trie
    for (int i = 0; i < n; i++)
        insert(root, keys[i]);

    // Search for different keys
    search(root, "the")? cout << "Yes\n" :
                         cout << "No\n";
    search(root, "these")? cout << "Yes\n" :
                           cout << "No\n";
    return 0;
}
```

Graphs

Strengths: represents links, works well with connections
Weakness: scaling challenge because most graphs are O(nlogn) or slower

Directed/undirected: direction vs no direction

Implementation
Could be represented by:

1. Edge list: a list of all edges of each int
    vector <vector<int>> = {{0,1},{1,2},{1,3}};
2. Adjacency list: index represents node, value is a list of all neighbors
    vector<vector<int>> or unordered map of lists/vector<ints>
3. Adjacency matrix: a matrix of 0s and 1s indicating whether node x connects to node y

Boom filter

A Bloom filter is a data structure designed to tell you, rapidly and memory-efficiently, whether an element is present in a set.

The price paid for this efficiency is that a Bloom filter is a probabilistic data structure: it tells us that the element either *definitely is not* in the set or *may be* in the set.

C++ Syntax

sort(first, last) - Sorts the elements in the range [first,last) into ascending order.

Strings
string subbed = stringName.substr(pos1,pos2);
string stringToFind = stringName.find("string I'm looking for");

Looping:

```
for(char& c : str) { do_things_with(c); }
```

```
for(std::string::size_t i = 0; i < str.size(); ++i)
```

Vectors
vector<varType> vectorName;
Iterators: use below

Modifiers:
assign() – It assigns new value to the vector elements by replacing old ones
push_back() – It push the elements into a vector from the back
pop_back() – It is used to pop or remove elements from a vector from the back.
insert() – It inserts new elements before the element at the specified position
erase() – It is used to remove elements from a container from the specified position or range.
swap() – It is used to swap the contents of one vector with another vector of same type and size.
clear() – It is used to remove all the elements of the vector container
emplace() – It extends the container by inserting new element at position
emplace_back() – It is used to insert a new element into the vector container, the new element is added to the end of the vector

Iterators
template <class InputIterator, class T>
  InputIterator find (InputIterator first, InputIterator last, const T& val);

vector<int> array;
vector<int>::iterator ptr;
for (ptr = array.begin(); ptr < array.end; ptr++)

Unordered sets
unordered_set<valueType> setName;
setName.insert(value);
setName.find(key); //index
setName.begin() // iterator to first element
setName.end() //iterator to last element

Unordered maps
unordered_map<keyType,valueType> mapName
To insert: mapName(key) = value;

- at() : This function in C++ unordered_map returns the reference to the value with the element as key k.
- begin() : Returns an iterator pointing to the first element in the container in the unordered_map container
- end() : Returns an iterator pointing to the position past the last element in the container in the unordered_map container
- bucket() Returns the bucket number where the element with the key k is located in the map.
- bucket_count bucket_count is used to count the total no. of buckets in the unordered_map. No parameter is required to pass into this function.
- bucket_size Returns number of elements in each bucket of the unordered_map.
- iterator find (const key_type& k);
  const_iterator find (const key_type& k) const;

- ○ Searches the container for an element with k as key and returns an iterator to it if found, otherwise it returns an

Maps
Ordered by default, built on BST instead of hash table,
map<keyType,valueType> mapName

- begin() – Returns an iterator to the first element in the map
- end() – Returns an iterator to the theoretical element that follows last element in the map
- size() – Returns the number of elements in the map
- max_size() – Returns the maximum number of elements that the map can hold
- empty() – Returns whether the map is empty
- pair insert(keyvalue,mapvalue) – Adds a new element to the map
- erase(iterator position) – Removes the element at the position pointed by the iterator
- erase(const g)– Removes the key value ‘g’ from the map
- clear() – Removes all the elements from the map