## Neat Code

#106 Construct Binary Tree from Inorder and Postorder Traversal: idx/iter to indicate array range instead of creating subarray.

#79 Word Search: neat code

#54 Spiral Matrix: iteration instead of recursion.

#18 4Sum: extra condition judgement to speed up your code; duplicate dealing

#355 Design Twitter: use algorithm/data structure knowledge to solve real problems.

#347 Top K Frequent Elements: classic map and priority queue problem.

#166 Fraction to Recurring Decimal: carefully consider all the situations including negative number. Get clear by reducing real example

#24 Swap Nodes in Pairs : consider the body first, write loop conditions after

# 92 Reverse Linked List II : more neat code, draw picture to help understand the situation

#29 Divide Two Integers: bit operation as multiplication.

# Sqrt(x)

#46 Permutations: give clear definition to your variable and method!!!!

## Two Pointers

no matter it is window problem or else, if you need to find two pointers in array, you can move two pointers from the same side or from two sides.

## Min/Max Window

1. **Use** two pointers: **start** **and** **end** **to** represent a window.

2. **Move** **end** **to** find a valid window.

3. **When** a valid window **is** **found**, **move** **start** **to** find a smaller window.

you should use other variables like array or map to help decide whether a window is valid.

take care where to update the min/max.

**int** **findSubstring**(string s){

vector<**int**> map(128,0);

**int** counter; *// check whether the substring is valid*

**int** begin=0, end=0; *//two pointers, one point to tail and one head*

**int** d; *//the length of substring*

**for**() { */\* initialize the hash map here \*/* }

**while**(end<s.size()){

**if**(map[s[end++]]-- ?){ */\* modify counter here \*/* }

**while**(*/\* counter condition \*/*){

***/\* update d here if finding minimum\*/***

*//increase begin to make it invalid/valid again*

**if**(map[s[begin++]]++ ?){ */\*modify counter here\*/* }

}

***/\* update d here if finding maximum\*/***

}

**return** d;

class Solution {

public:

int minSubArrayLen(int s, vector<int>& nums) {

int N = nums.size();

int i = 0;

int j = 0;

int sum = 0;

int minLen = INT32\_MAX;

while(j < N){

while(j < N && sum < s){

sum += nums[j];

j++;

}

if(sum < s){

break;

}

while(sum >= s){

sum -= nums[i];

i++;

}

minLen = min(minLen, j - i + 1);

}

return minLen == INT32\_MAX ? 0 : minLen;

}

};

class Solution {

public:

int lengthOfLongestSubstring(string s) {

int N = s.size();

int lo = 0;

int hi = 0;

vector<int> idx(256, 0);

int maxLen = 0;

while(hi < N)

{

idx[s[hi]]++;

while(idx[s[hi]] > 1)

{

idx[s[lo]]--;

lo++;

}

maxLen = max(maxLen, hi - lo + 1);

hi++;

}

maxLen = max(maxLen, hi - lo);

return maxLen;

}

};

class Solution {

public:

string minWindow(string s, string t) {

map<char, int> dic;

int N = s.size();

int i = 0;

int j = 0;

int minLen = INT32\_MAX;

int start = 0;

int count = t.size();

for(auto c: t)

{

dic[c]++;

}

while(j < N)

{

if(dic.find(s[j])==dic.end())

{

j++;

continue;

}

if(dic[s[j]] > 0)

{

count--;

}

dic[s[j]]--;

while(count == 0)

{

if(j-i+1 < minLen)

{

minLen = j-i+1;

start = i;

}

if(dic.find(s[i])!=dic.end())

{

dic[s[i]]++;

if(dic[s[i]] > 0)

{

count = 1;

}

}

i++;

}

j++;

}

return minLen == INT32\_MAX ? "" : s.substr(start, minLen);

}

};

## Backtracking

Restor IP Address:

add additional check first to avoid unnecessray situation.

End loop/recursion in advance

class Solution {

public:

vector<vector<int>> combinationSum(vector<int>& candidates, int target) {

vector<vector<int>> ret;

vector<int> ans;

sort(candidates.begin(),candidates.end());

helper(candidates, target, 0, ans, ret);

return ret;

}

void helper(vector<int>& candidates, int target, int start, vector<int>& ans, vector<vector<int>>& ret)

{

if(target == 0) **// modify target in each recursion until finding an answer.**

{

ret.push\_back(ans);

return;

}

for(int i = start; **i < candidates.size() && candidates[i]<=target**; i++)

{

ans.push\_back(candidates[i]);

helper(candidates, target-candidates[i], i, ans, ret);

ans.pop\_back();

}

}

};

class Solution {

public:

vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {

vector<vector<int>> ret;

vector<int> ans;

sort(candidates.begin(), candidates.end());

helper(candidates, target, 0, ans, ret);

return ret;

}

void helper(vector<int>& candidates, int target, int start, vector<int>& ans, vector<vector<int>>& ret)

{

if(target==0)

{

ret.push\_back(ans);

return;

}

for(int i = start; **i < candidates.size() && candidates[i] <= target**; i++)

{

**if(i == start || candidates[i] != candidates[i-1])**

{

ans.push\_back(candidates[i]);

helper(candidates, target-candidates[i], i+1, ans, ret);

ans.pop\_back();

}

}

}

};

public:

vector<vector<int>> combinationSum3(int k, int n) {

vector<vector<int>> ret;

vector<int> ans;

helper(k, n, ans, ret); // **modify k and n in each recursion to find an answer.**

return ret;

}

void helper(int k, int n, vector<int>& ans, vector<vector<int>>& ret)

{

if(k == 0 && n == 0)

{

ret.push\_back(ans);

return;

}

if(k > 0)

{

for**(int i = ans.size()==0 ? 1 : ans.back()+1; i <= 9 && i <= n**; i++)

{

ans.push\_back(i);

helper(k-1, n-i, ans, ret);

ans.pop\_back();

}

}

}

};

## Binary Search

If you want to find a number in a range which santisfies a condition,

you can use binary search.

However, to speech up : narrow the range ahead.

x: 1, 2, …., 2^(n), 2^(n+1)…

so that it’s in the range [2^(n), 2^(n+1)

## Dynamic Programming

**do back-up** to avoid duplicate computation!!!

**c++ pass vector by reference, and pass class by pointer**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* public int val;

\* public TreeNode left;

\* public TreeNode right;

\* public TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public IList<TreeNode> GenerateTrees(int n)

{

if(n ==0 )

{

return new List<TreeNode>();

}

return GenerateTreesHelper(1, n);

}

public static Dictionary<Tuple<int, int>, IList<TreeNode>> dic = new Dictionary<Tuple<int, int>, IList<TreeNode>>();

private static IList<TreeNode> GenerateTreesHelper(int lo, int hi)

{

if(dic.ContainsKey(new Tuple<int, int>(lo, hi)))

{

return dic[new Tuple<int, int>(lo, hi)];

}

var ret = new List<TreeNode>();

if(lo>hi)

{

ret.Add(null);

return ret;

}

for(var i = lo;i<= hi;i++)

{

var left = GenerateTreesHelper(lo, i - 1);

var right = GenerateTreesHelper(i + 1, hi);

foreach(var l in left)

{

foreach(var r in right)

{

var root = new TreeNode(i);

root.left = l;

root.right = r;

ret.Add(root);

}

}

}

dic.Add(new Tuple<int, int>(lo, hi), ret);

return ret;

}

}

## Object-oriented Programming

Design the API frist without considering the under-lying data structure

Hanoi Tower Problem

## Bit Operation

pow(2, i) => 1 << i

use bit operation for times

## Tips for C++

1\ reference and pointer as parameter: if you pass vector<> map<> set<> self-defined class as parameter

2\ use pointer: arrow instead of dot

## Brain Teasers

**Examplify, Simplify and Generalize(make x , y stand for specify number), Pattern Matching and Base Case and Build can be especially helpful.**

From high level to low level(examplify, base case and build), suppose and make assumption

From low level to high level(simplify and generalize)

suppose there are XXX

suppose it is like this XXX

## Math and Probability

**Ask questions**. This question has a lot of unknowns—ask questions to clarify them.

Many interviewers intentionally ask vague questions to see if you'll clarify your

assumptions.

• **When possible, design and use data structures**. It shows that you understand and

care about object-oriented design.

• Think through which data structures you design to represent a line.**There are a lot of**

**options, with lots of trade-offs. Pick one, and explain your choice.**

• **Don't assume that the slope and y-intercept are integers.**

• **Understand limitations of floating point representations. Never check for equality**

**with ==. Instead, check if the difference is less than an epsilon value.**

When you get this question, do your best to solve it—even though it's really difficult.

You can start with a brute force approach (challenging, but not quite as tricky), and then

you can start trying to optimize it. Or, try to find a pattern in the numbers.

Chances are that your interviewer will help you along when you get stuck. Whatever you

do, don't give up! Think out loud, wonder out loud, and explain your thought process.

Your interviewer will probably jump in to guide you.

Remember, perfection on this problem is not expected. Your performance is evaluated

in comparison to other candidates. Everyone struggles on a tricky problem.

## Object-oriented Design

make assumption to clarify and simplify the problem. Ask the interviewer if it is ok blablabla.

The main function your system should provide.

The main object your system should include.

**How to Approach Object-Oriented Design Questions**

Regardless of whether the object is a physical item or a technical task, object-oriented

design questions can be tackled in similar ways. The following approach will work well

for many problems.

***Step 1: Handle Ambigulty***

Object-oriented design (OOD) questions are often intentionally vague in order to test

whether you'll make assumptions or if you'll ask clarifying questions. After all, a developer

who just codes something without understanding what she is expected to create

wastes the company's time and money, and may create much more serious issues.

When being asked an object-oriented design question, you should inquire ***who* is going**

**to use it and *how* they are going to use it.** Depending on the question, you may even

want to go through the "six Ws": who, what, where, when, how, why.

For example, suppose you were asked to describe the object-oriented design for a

coffee maker. This seems straightforward enough, right? Not quite.

Your coffee maker might be an industrial machine designed to be used in a massive

restaurant servicing hundreds of customers per hour and making ten different kinds

of coffee products. Or it might be a very simple machine, designed to be used by the

elderly for just simple black coffee. These use cases will significantly impact your design.

***Step 2: Define the Core Objects***

Now that we understand what we're designing, we should consider what the "**core**

**objects" in a system** are. For example, suppose we are asked to do the object-oriented

design for a restaurant. Our core objects might be things like Table, Guest, Party,

Order, Meal, Employee, Server, and Host.

***Step 3: Analyze Relationships***

Having more or less decided on our core objects, **we now want to analyze the relationships**

**between the objects.** Which objects are members of which other objects? Do any

objects inherit from any others? Are relationships many-to-many or one-to-many?

For example, in the restaurant question, we may come up with the following design:

• Party should have an array of Guests.

• Server and Host inherit from Employee.

• Each Table has one Party, but each Party may have multiple Tables.

• There is one Host for the Restaurant.

Be very careful here—you can often make incorrect assumptions. For example, a single

Table may have multiple Parties (as is common in the trendy "communal tables" at

some restaurants). You should talk to your interviewer about **how general purpose your**

**design should be.**

***Step 4: Investigate Actions***

At this point, **you should have the basic outline of your object-oriented design**. What

remains is to consider the key actions that the objects will take and how they relate to

each other. You may find that you have forgotten some objects, and you will need to

update your design.

For example, a Party walks into the Restaurant, and a Guest requests a Table from

the Host. The Host looks up the Reservation and, if it exists, assigns the Party to a

Table. Otherwise, the Party is added to the end of the list. When a Party leaves, the

Table is freed and assigned to a new Party in the list.

Make decision based on whether the system is big or small. In a big system, we may want to develop different component seperately to achieve maintaince and expansionbility, flexibility. In a small system, it may makes things complicated but still worth a try.

I’ll try to sketch out the code.

8.1 Deck of cards and BlackJack

objects: card, blackjack card(inherited from card), player(list of cards, able to get score), deck of cards

8.2 Call Center

1\ Analyse the real situation. How does it really work. There should be multiple call to deal with than one call.

2\ main object: call, employee(respondent, manager, director), call center

3\ the data structure you choose to deal with problem: queue, stack, priority-queue

4\ ignore the algorithm and implementation first, think about the object, method, API, function that you need.

Employee e = dispatch(call);

//return null if not dispatched otherwise returns the first available employee.

## Recursion and DP

When you hear a problem beginning with the following statements, it's often (though

not always) a good candidate for recursion: "Design an algorithm to compute the nth

,..";"Write code to list the first n...";"Implement a method to compute all..."; etc..

Practice makes perfect! The more problems you do, the easier it will be to recognize

recursive problems.

**How to Approach**

Recursive solutions, by definition, are built off solutions to sub-problems. Many times,

this will mean simply to compute f (n) by adding something, removing something,

or otherwise changing the solution for f (n-1). In other cases, you might have to do

something more complicated.

You should consider both bottom-up and top-down recursive solutions. The Base Case

and Build approach works quite well for recursive problems.

*Bottom-Up Recursion*

Bottom-up recursion is often the most intuitive. We start with knowing how to solve the

problem for a simple case, like a list with only one element, and figure out how to solve

the problem for two elements, then for three elements, and so on. The key here is to

think about how you can *build the* solution for one case off of the previous case.

*Top-Down Recursion*

Top-Down Recursion can be more complex, but it's sometimes necessary for problems.

In these problems, we think about how we can divide the problem for case N into subproblems.

Be careful of overlap between the cases.

CrackingTheCodinglnterview.

**Dynamic Programming**

Dynamic programming (DP) problems are rarely asked because, quite simply, they're

too difficult for a 45-minute interview. Even good candidates would generally do so

poorly on these problems that it's not a good evaluation technique.

If you're unlucky enough to get a DP problem, you can approach it much the same way

as a recursion problem.The difference is **that intermediate results are"cached"for future**

**calls.**

Recursion VS. Iteration

recursion puts heavy load on memory. It may cause stack overflow, because every time you call the recursion function it adds a new layer in stack.

## Scalability and Memory Limits

**Interviewers are not trying to test your knowledge of system design; in fact, interviewers**

**rarely try to test knowledge of anything but the most basic Computer Science concepts.**

**Instead, they are evaluating your ability to break down a tricky problem and to solve**

**problems using what you do know**. The following approach works well for many system

design problems.

*Step 1: Make Believe*

**Pretend that the data can all fit on one machine and there are no memory limitations.**

**How would you solve the problem?**This answer to this question will provide the general

outline for your solution.

*Step 2: Get Real*

Now, go back to the original problem. How much data can you fit on one machine,

and **what problems will occur when you split the data up?** Common problems include

figuring out how to logically divide the data up, and how one machine would identify

where to look up a different piece of data.

*Step 3: Solve Problems*

Finally, think about how to solve the issues you identified in Step 2. Remember that the

solution for each issue might be to actually remove the issue entirely, or it might be

to simply mitigate the issue. Usually, you can continue to use (with modifications) the

approach you outlined in Step 1, but occasionally you will need to fundamentally alter

the approach.

*Dividing Up Lots of Data*

Though we can sometimes increase hard drive space in a computer, there comes a point

where data simply must be divided up across machines.The question, then, is what data

belongs on which machine? There are a few strategies for this.

• *By Order of Appearance:*

We could simply divide up data by order of appearance. That is, as new data comes

in, we wait for our current machine to fill up before adding an additional machine.

This has the advantage of never using more machines than are necessary. However,

depending on the problem and our data set, our lookup table may be more complex

and potentially very large.

• *By Hash Value:*

An alternative approach is to store the data on the machine corresponding to the

hash value of the data. More specifically, we do the following: (1) pick some sort of

*key* relating to the data, (2) hash the key, (3) mod the hash value by the number of

machines, and (4) store the data on the machine with that value. That is, the data is

stored on machine #[mod(hash(key) j N)].

The nice thing about this is that there's no need for a lookup table. Every machine

will automatically know where a piece of data is. The problem, however, is that a

machine may get more data and eventually exceed its capacity. In this case, we may

need to either shift data around the other machines for better load balancing (which

is very expensive), or split this machine's data into two machines (causing a tree-like

structure of machines).

• *By Actual Value:*

Dividing up data by hash value is essentially arbitrary; there is no relationship

between what the data represents and which machine stores the data. In some

cases, we may be able to reduce system latency by using information about what

the data represents.

For example, suppose you're designing a social network. While people do have

friends around the world, the reality is that someone living in Mexico will probably

have a lot more friends from Mexico than an average Russian citizen. We could,

perhaps, store "similar" data on the same machine so that looking up the Mexican

person's friends requires fewer machine hops.

Before we start solving the problem, we need to understand whether this is a one time

only operation, or if this findWords procedure will be called repeatedly. Let's assume

that we will be calling findWords many times for the same set of documents, and we

can therefore accept the burden of pre-processing.