**Moderate Question**

**Maximum Sub Array**

Question: Given an array with at least one positive integer, find the contiguous sub-array with the largest sum. For example, for the sequence of values −2, 1, −3, 4, −1, 2, 1, −5, 4; the contiguous sub-array with the largest sum is 4, −1, 2, 1, with sum 6.

public static int[] maxSubArray(int[] array) {

int starting\_index = 0;

int length = 0;

int sum = 0;

int t\_starting\_index = 0;

int t\_length = 0;

int t\_sum = 0;

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

if (t\_sum + array[i] > 0) {

t\_sum += array[i];

t\_length++;

if (t\_sum > sum) {

sum = t\_sum;

length = t\_length;

starting\_index = t\_starting\_index;

}

} else {

t\_sum = 0;

t\_length = 0;

t\_starting\_index = i + 1;

}

}

int[] sub = new int[length];

System.arraycopy(array, starting\_index, sub, 0, length);

return sub;

}

Solution to find the max sum: [Kadane Algorithm]

public static int maxSubArray(int[] array) {

int max\_ending\_here = 0;

int max\_so\_far = 0;

for (int i: array) {

max\_ending\_here = Math.max(0, max\_ending\_here + i);

max\_so\_far = Math.max(max\_so\_far, max\_ending\_here);

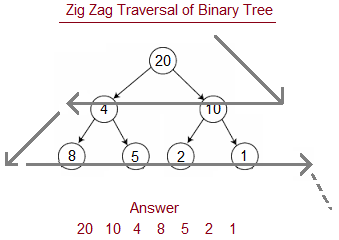
}

return max\_so\_far;

}

**Zig-Zag Tree Traversal**

Question: Print out items of a Binary Tree in a zig-zag fashion.

​

public static void zigZag(Node root) {

Stack < Node > s1 = new Stack < Node > ();

Stack < Node > s2 = new Stack < Node > ();

s1.push(root);

while (!s1.isEmpty() || !s2.isEmpty()) {

while (!s1.isEmpty()) {

Node n = s1.pop();

System.out.println(n.data);

if (n.right != null) s2.push(n.right);

if (n.left != null) s2.push(n.left);

}

while (!s2.isEmpty()) {

Node n = s2.pop();

System.out.println(n.data);

if (n.left != null) s1.push(n.left);

if (n.right != null) s1.push(n.right);

}

}

}

**Find nearest 100 points**

 Find the 100 points closest to the origin in a 2D graph.

**Focus**

Complexity analysis, heap data structures or binary search

**Solution**

Read each element. Calculate its distance from the origin. If the set had fewer than 100 elements, add it. If it has more than 100 elements, but is smaller than the largest element, remove the largest element, add the new smaller element. Repeat until all points have been accounted for.

This one is fairly straight forward. You’ll need to maintain an ordered list of some type, access it intelligently and prune it as necessary. Now, the easiest way to do this is to use a max heap and remove an element for every element over the 100th that’s added. However, specific knowledge about max heaps isn’t required. You could do the same thing with an ordered list of times that’s bounded at a size of 100 elements. Any solution that runs in O(n lg m) time is acceptable. I’d even accept a solution that runs in O(nm) time if they can justify that the m is a known constant. Obviously, the candidate will need to also be able to calculate the distance from the origin. If you want a fun twist you can tell the candidate that they don’t have access to any math packages and they can only use addition, multiplication, division, and subtraction. Note: We just want the 100 points closest to the origin; since we don’t ever actually have to calculate the actual distance from the origin (just the relative distance), you don’t need a square root function to solve this problem.

**Common Mistakes**

There are some candidates that don’t understand the problem and will try to use O(n) space instead of O(100) space. They should be told that optimising for space here also helps to optimise for time. Some candidates get hung up on calculating the actual distance from the origin and use some new invented math. I’m fine with anything that actually works here, but ask candidates to explain their design decisions.

**Analysis**

Again, we’re simply looking for candidates to get to a solution that runs in a reasonable amount of time. If they can get to the O(n) solutions to this problem with a reasonable amount of guidance they’ve passed the question. If they can get the solution, but not do the runtime analysis, you’ll need to make a call as to whether they’re just have problems with runtimes on this problem or if there’s a serious gap somewhere in their knowledge. We shouldn’t bring any candidate in house that can’t do some level of runtime analysis (they don’t need to be absolutely perfect, but close to perfect is desirable). If they can’t get to a solution that runs in the right time or space they’ve failed the question.

**Pair of numbers summing to a number k**

Allow time: 20-30 minutes

Suitable for: 1st phone screen

Look for: Ability to apply computer fundamentals, data structures, time complexity, ability to compare between multiple approaches to solving a problem

Given a list of numbers and another number k, find IF there are any 2 numbers whose sum is k (Please note that we are only finding IF a pair exists and we are not trying to find all pairs here).

Some of the questions that candidates ask are:

1) Are negative numbers allowed ? Yes

This is a easy problem but there are multiple solutions to this problem:

1. Brute force: Have nested for loops. This is a good start and almost all candidates get this. If they get this, ask about time complexity of this solution which is O(n square) If they give me this solution, I ask them to improve the time complexity of this solution.

2. Sorting and then do a binary search: Once you sort the list the list, you are looking for k-a[i] for each number in list. Hence, complexity is O(nlogn).

3. Another possible solution is to sort the list and then have 2 pointers. One from start of the list and other from end of the list. if a[start]+a[end] == k, you are done. If a[start]+a[end] > k, --end. If If a[start]+a[end] < k, ++ start. The complexity of this is O(nlogn).

Some candidates are smart and tell this solution in first go. That is good. In that case, I ask them to improve the time complexity of this solution and not worry about space.

4. Then there is Hashset solution. It is extremely important about how you insert numbers into Hashset. A lot of candidates say that they will insert all numbers into hashmap. Then they will start from 0 and check if k-a[i] exists. This approach ha one edge where it will give false positives if k = 2\*a[i]. Let us say that list has only 1 element and that element is 2. Assume k = 4. If we follow this approach, it will give us false positive. Then some candidates will say that they will just filter on size of list. This is again incorrect because you can still have scenario where list is 3,2,9 and k = 18. This approach will still give false positives. Hence, we need to see how candidates solves this. Best way is to use HashSet and look into hashset for k-a[i] before inserting a[i] into hashset.

Some candidates use HashMap instead of hashset. In that case, please ask candidates what are they inserting into key and value for HashMap.

After finishing this question, I generally ask candidate to code up this hashset solution and email me in 10 min (while I am still on phone).

The Min Work Time Problem

Allow time: 40+ minutes

Look for: ability to understand and communicate, ability to deal with setbacks, ability to convert thought into code, rudimentary understanding of time complexity.

Problem Statement:

There are n tasks, each with associated integer valued processing time ti 1 ≤ i ≤ n. There is a pool of m worker threads, m ≤ n. Design an algorithm that finds an allocation of tasks to threads that minimizes the processing time (as defined by longest processing time of any thread) subject to the following constraint:

Allocation of tasks to threads must be consecutive per thread. For example for tasks 1, 2, 3, 4 and threads A and B. Allocating 1, 2 to A and 3, 4 to B is legal. But 1, 3 to A, 2, 4 to B is not.

Problem instance encoding:

A string consists of n+1 integer values separated by " ". The first number is the value m, followed by n numbers that denotes the processing time of each task.

Example: "4 3 1 5 6 4" denotes a problem instance where there are 4 processing units and 5 tasks with processing time 3, 1, 5, 6, and 4

Solution encoding: A number that represents the minimum processing time. Example: for the sample problem, the optimal solution is encoded as "6"

I think this problem came originally from a Top Coder tournament. Normally I don't expect a candidate to reach the optimal solution in a phone screen or in-house interview. You should use this problem to interact with candidates and see how they approach the problem, and provide counter examples to show them that they are wrong, and see how they react and refine their approaches. With enough iterations (2-3 failed attempts), you can let the candidate just pick a not-perfect solution and implement it to check his/her coding skill.

To use this problem successfully, you need to be prepared to provide counter examples for a variety of "heuristics".

[*TODO: Provide a list of heuristics and counter examples*]

One good solution is to transform the problem into a decision problem: is there a solution that achieves processing time x? This decision problem can be solved greedily in O(n). We can repeatedly ask this question and perform binary search for a solution in the solution space. The solution must be at least max{Ti: 1 ≤ i ≤ n} (cannot be less than the most expensive task), and at most &Sigma;i = (1 .. n) Ti (assign all the tasks to 1 worker). The time complexity is thus

O(n \* log(&Sigma;i = (1 .. n) Ti - max{Ti: 1 ≤ i ≤ n})) ⊆ O(n \* log(&Sigma;i = (1 .. n) Ti)).

There's also another dynamic programming solution which was coded up by Victor. If you know how the DP solution works please let me know.

koujalag@ : Binary search ?

Min possible time is 0 (or max time among all when we have enough threads to do all at the same time). Max would be sum of times of all the tasks ( if there were a sequential assignment ).

Since the assignment is order specific, (1 2 A), we can binary search over 0..sum of all times.

For each "mid" value, iterate through the array and see where do we want to break for each thread. (that is, when time exceeds mid value).

Finally, see if it possible to finish with m threads. If yes, hi = mid-1, else, lo = mid + 1.

Time complexity would be O(number of array elements \* log ( sum of all times ) )

**How Much Water Can A Bar Graph Hold**

Allow time: 20-30 minutes

Suitable for: 2nd phone screen and in-house, might be good for a 1st PS take-home.

Look for: ability to understand and communicate, ability to convert thought into code, rudimentary understanding of time complexity.

Problem Statement:

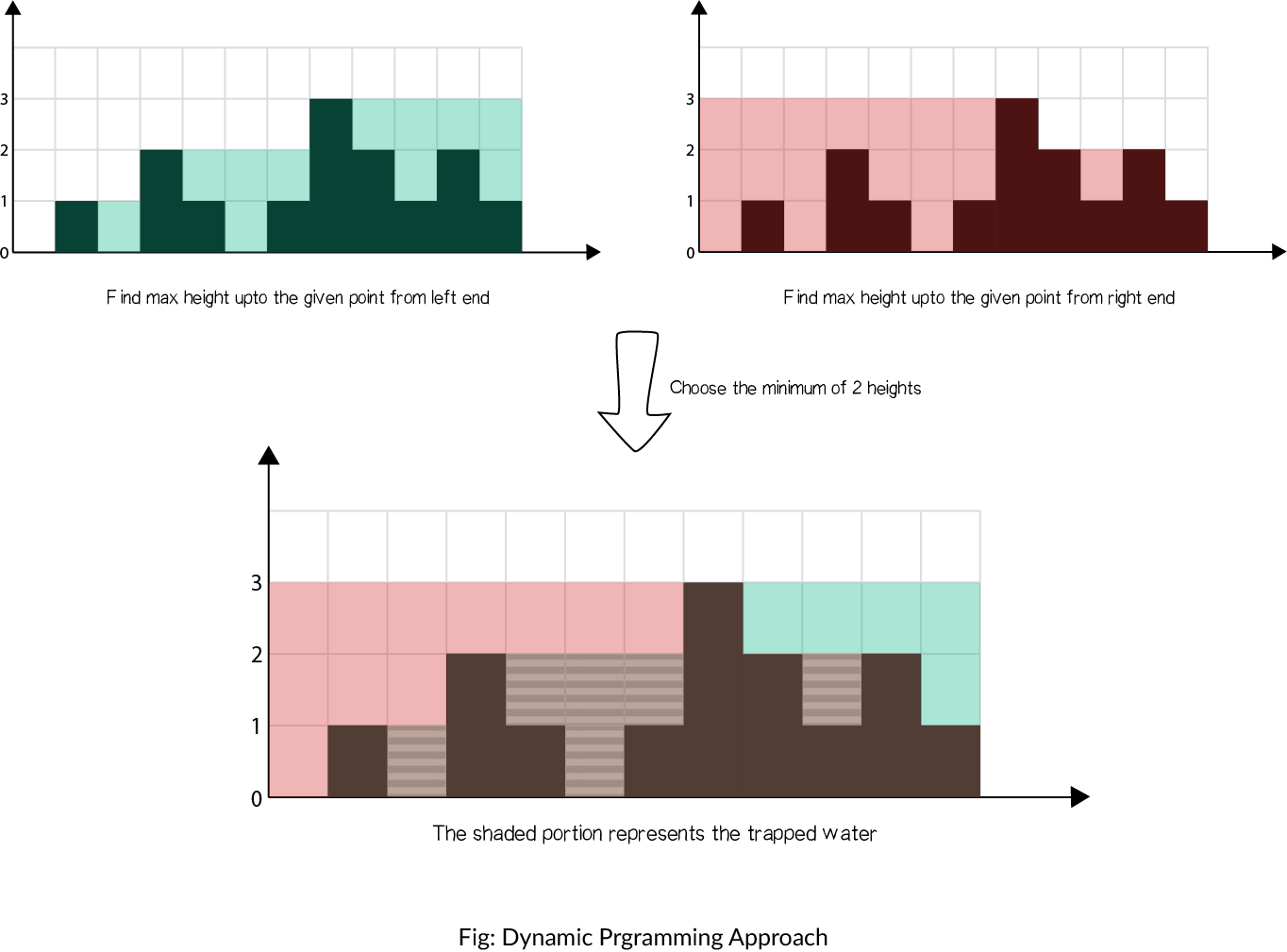
Given an bar graph, encoded as an array of non-negative integers, calculate the units of water that this bar graph can hold.

**Approach 1: Dynamic Programming**

**Intuition**

In brute force, we iterate over the left and right parts again and again just to find the highest bar size upto that index. But, this could be stored. Voila, dynamic programming.

The concept is illustrated as shown:

​

**Algorithm**

* Find maximum height of bar from the left end upto an index i in the array \text{left\\_max}left\_max.
* Find maximum height of bar from the right end upto an index i in the array \text{right\\_max}right\_max.
* Iterate over the \text{height}height array and update ans:
  + Add \min(\text{max\\_left}[i],\text{max\\_right}[i]) - \text{height}[i]min(max\_left[i],max\_right[i])−height[i] to ansans

**Approach 2: Using stacks**

**Intuition**

Instead of storing the largest bar upto an index as in [Approach 2](https://leetcode.com/problems/trapping-rain-water/solution/#approach-2-dynamic-programming), we can use stack to keep track of the bars that are bounded by longer bars and hence, may store water. Using the stack, we can do the calculations in only one iteration.

We keep a stack and iterate over the array. We add the index of the bar to the stack if bar is smaller than or equal to the bar at top of stack, which means that the current bar is bounded by the previous bar in the stack. If we found a bar longer than that at the top, we are sure that the bar at the top of the stack is bounded by the current bar and a previous bar in the stack, hence, we can pop it and add resulting trapped water to \text{ans}ans.

**Algorithm**

* Use stack to store the indices of the bars.
* Iterate the array:
  + While stack is not empty and \text{height}[current]>\text{height}[st.top()]height[current]>height[st.top()]
    - It means that the stack element can be popped. Pop the top element as \text{top}top.
    - Find the distance between the current element and the element at top of stack, which is to be filled. \text{distance} = \text{current} - \text{st.top}() - 1distance=current−st.top()−1
    - Find the bounded height \text{bounded\\_height} = \min(\text{height[current]}, \text{height[st.top()]}) - \text{height[top]}bounded\_height=min(height[current],height[st.top()])−height[top]
    - Add resulting trapped water to answer \text{ans} \mathrel{+}= \text{distance} \times \text{bounded\\_height}ans+=distance×bounded\_height
  + Push current index to top of the stack
  + Move \text{current}current to the next position

**Complexity analysis**

* Time complexity: O(n)O(n).
  + Single iteration of O(n)O(n) in which each bar can be touched at most twice(due to insertion and deletion from stack) and insertion and deletion from stack takes O(1)O(1) time.
* Space complexity: O(n)O(n). Stack can take upto O(n)O(n) space in case of stairs-like or flat structure.

**Approach 3: Using 2 pointers**

**Intuition**

As in [Approach 2](https://leetcode.com/problems/trapping-rain-water/solution/#approach-2-dynamic-programming), instead of computing the left and right parts seperately, we may think of some way to do it in one iteration. From the figure in dynamic programming approach, notice that as long as \text{right\\_max}[i]>\text{left\\_max}[i]right\_max[i]>left\_max[i] (from element 0 to 6), the water trapped depends upon the left\_max, and similar is the case when \text{left\\_max}[i]>\text{right\\_max}[i]left\_max[i]>right\_max[i] (from element 8 to 11). So, we can say that if there is a larger bar at one end (say right), we are assured that the water trapped would be dependant on height of bar in current direction (from left to right). As soon as we find the bar at other end (right) is smaller, we start iterating in opposite direction (from right to left). We must maintain \text{left\\_max}left\_max and \text{right\\_max}right\_max during the iteration, but now we can do it in one iteration using 2 pointers, switching between the two.

**Algorithm**

* Initialize \text{left}left pointer to 0 and \text{right}right pointer to size-1
* While \text{left}< \text{right}left<right, do:
  + If \text{height[left]}height[left] is smaller than \text{height[right]}height[right]
    - If \text{height[left]} \geq \text{left\\_max}height[left]≥left\_max, update \text{left\\_max}left\_max
    - Else add \text{left\\_max}-\text{height[left]}left\_max−height[left] to \text{ans}ans
    - Add 1 to \text{left}left.
  + Else
    - If \text{height[right]} \geq \text{right\\_max}height[right]≥right\_max, update \text{right\\_max}right\_max
    - Else add \text{right\\_max}-\text{height[right]}right\_max−height[right] to \text{ans}ans
    - Subtract 1 from \text{right}ri​

1 / 11

**Complexity analysis**

* Time complexity: O(n)O(n). Single iteration of O(n)O(n).
* Space complexity: O(1)O(1) extra space. Only constant space required for \text{left}left, \text{right}right, \text{left\\_max}left\_max and \text{right\\_max}right\_max

**Merge k Sorted Lists into One Sorted List**

Allow time: 40-50 minutes

Look for: Good understanding of algorithm and complexity. Ability to convert thought into code.

This problem very easy to understand. One way is to maintain a heap of the k lists and continue to draw from the top of the heap (and to adjust the heap). This method achieves O(log(k) \* k \* n), assuming average list length is n. Beware that it requires no more than O(k) extra memory, besides the memory required to hold input and output. If you consider the k sorted list as input streams, and output also goes to an output stream, no memory is even required to hold input/output.

Here's a ruby solution that uses a heap and the iterator function pattern, creating an iterator for each list, and constructing a master iterator that abstract away the heap. Due to the length of the solution, you may want to just let the candidate assume that there's a priority queue class ready to be used in code, and then ask about the implementation of the priority queue (how to achieve log(k) dequeue)

// C++ implementation to merge k sorted linked lists

// | Using MIN HEAP method

#include <bits/stdc++.h>

using namespace std;

struct Node {

int data;

struct Node\* next;

};

// 'compare' function used to build up the

// priority queue

struct compare {

bool operator()(struct Node\* a, struct Node\* b)

{

return a->data > b->data;

}

};

// function to merge k sorted linked lists

struct Node\* mergeKSortedLists(struct Node\* arr[], int k)

{

struct Node\* head = NULL, \*last;

// priority\_queue 'pq' implemeted as min heap with the

// help of 'compare' function

priority\_queue<Node\*, vector<Node\*>, compare> pq;

// push the head nodes of all the k lists in 'pq'

for (int i = 0; i < k; i++)

pq.push(arr[i]);

// loop till 'pq' is not empty

while (!pq.empty()) {

// get the top element of 'pq'

struct Node\* top = pq.top();

pq.pop();

// check if there is a node next to the 'top' node

// in the list of which 'top' node is a member

if (top->next != NULL)

// push the next node in 'pq'

pq.push(top->next);

// if final merged list is empty

if (head == NULL) {

head = top;

// points to the last node so far of

// the final merged list

last = top;

}

else {

// insert 'top' at the end of the merged list so far

last->next = top;

// update the 'last' pointer

last = top;

}

}

// head node of the required merged list

return head;

}

// function to print the singly linked list

void printList(struct Node\* head)

{

while (head != NULL) {

cout << head->data << " ";

head = head->next;

}

}

// Utility function to create a new node

struct Node\* newNode(int data)

{

// allocate node

struct Node\* new\_node = new Node();

// put in the data

new\_node->data = data;

new\_node->next = NULL;

return new\_node;

}

// Driver program to test above

int main()

{

int k = 3; // Number of linked lists

int n = 4; // Number of elements in each list

// an array of pointers storing the head nodes

// of the linked lists

Node\* arr[k];

// creating k = 3 sorted lists

arr[0] = newNode(1);

arr[0]->next = newNode(3);

arr[0]->next->next = newNode(5);

arr[0]->next->next->next = newNode(7);

arr[1] = newNode(2);

arr[1]->next = newNode(4);

arr[1]->next->next = newNode(6);

arr[1]->next->next->next = newNode(8);

arr[2] = newNode(0);

arr[2]->next = newNode(9);

arr[2]->next->next = newNode(10);

arr[2]->next->next->next = newNode(11);

// merge the k sorted lists

struct Node\* head = mergeKSortedLists(arr, k);

// print the merged list

printList(head);

return 0;

}

**Approach 1: Compare one by one**

**Algorithm**

* Compare every \text{k}k nodes (head of every linked list) and get the node with the smallest value.
* Extend the final sorted linked list with the selected nodes.

**Complexity Analysis**

* Time complexity : O(kN)O(kN) where \text{k}k is the number of linked lists.
  + Almost every selection of node in final linked costs O(k)O(k) (\text{k-1}k-1 times comparison).
  + There are NN nodes in the final linked list.
* Space complexity :
  + O(n)O(n) Creating a new linked list costs O(n)O(n) space.
  + O(1)O(1) It's not hard to apply in-place method - connect selected nodes instead of creating new nodes to fill the new linked list.

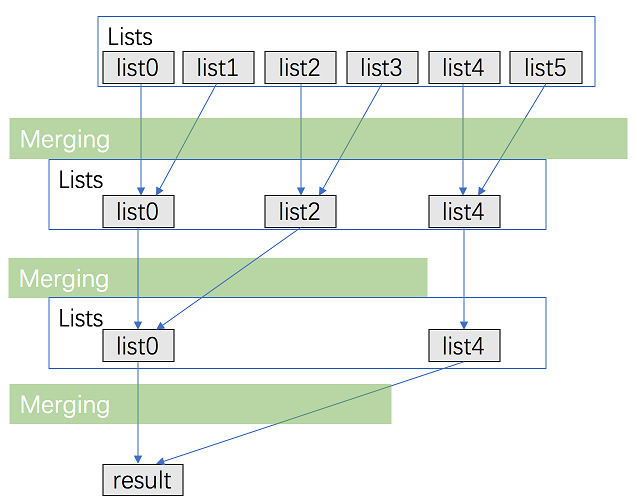
**Approach 2: Merge with Divide And Conquer**

**Intuition & Algorithm**

This approach walks alongside the one above but is improved a lot. We don't need to traverse most nodes many times repeatedly

* Pair up \text{k}k lists and merge each pair.
* After the first pairing, \text{k}k lists are merged into k/2k/2 lists with average 2N/k2N/k length, then k/4k/4, k/8k/8 and so on.
* Repeat this procedure until we get the final sorted linked list.

Thus, we'll traverse almost NN nodes per pairing and merging, and repeat this procedure about \log\_{2}{k}log2​k times.

​ {align = "center"}

**Moderate/Difficult Question**

**Find Duplicate in Array**

Given a read only array of n + 1 integers between 1 and n, find one number that repeats in linear time using less than O(n) space and traversing the stream sequentially O(1) times.

After this look for 2 duplicate element in N + 2 array and than multiple element to check problem ambiguity.

**Sample Input:**

[3 4 1 4 1]

**Sample Output:**

1

If there are multiple possible answers ( like in the sample case above ), output any one.

If there is no duplicate, output -1

**Solution Hint**

 Split the numbers from 1 to n in sqrt(n) ranges so that **range i corresponds to [sqrt(n) \* i .. sqrt(n) \* (i + 1)).**

Do one pass through the stream of numbers and figure out how many numbers fall in each of the ranges.

At least one of the ranges will contain more than sqrt(n) elements.

Do another pass and process just those elements in the oversubscribed range.

Using a **hash table** to keep frequencies, you’ll find a repeated element.

This is O(sqrt(n)) memory and 2 sequential passes through the stream

**Solution**

int Solution::repeatedNumber(const vector < int > & A) {

int i, xor1 = 0, xor2 = 1;

for (i = 0; i < A.size(); i++) {

xor1 ^= A[i];

}

int n = A.size();

for (i = 2; i <= (n - 1); i++) {

xor2 ^= i;

}

int repeat;

repeat = xor1 ^ xor2;

return repeat;

}

**N/3 Repeat Number**

You’re given a read only array of n integers. Find out if any integer occurs more than n/3 times in the array in linear time and constant additional space.

If so, return the integer. If not, return -1.

If there are multiple solutions, return any one.

**Example :**

Input : [1 2 3 1 1]

Output : 1

1 occurs 3 times which is more than 5/3 times.

**Solution Hint**

It works to simply pick all elements one by one. For every picked element, count its occurrences by traversing the array.  
If count becomes more than n/3, then print the element. **Time Complexity of this method would be O(n^2).**

A better solution is to use **sorting**.

First, sort all elements using a O(nLogn) algorithm. All required elements in a linear scan of array can be found once the array is sorted.

So overall, time complexity of this method is O(nLogn) + O(n) which is O(nLogn).

However, a linear solution is needed here.

*Note: if at any instance, you have three distinct elements from the array, if you remove them from the array, your answer does not change.*

Try to base your solution idea on the above fact.

Would it help to maintain two elements from array with their count as you traversed the array ?

**Solution**

It is important to note that if at a given time, you have 3 distinct element from the array, if you remove them from the array, your answer does not change.

Assume that we maintain 2 elements with their counts as we traverse along the array.

When we encounter a new element, there are 3 cases possible :

* We *don’t have 2 elements yet*. So add this to the list with count as 1.
* This element is *different from* the existing 2 elements. As we said before, we have 3 distinct numbers now. Removing them does not change the answer. So decrement 1 from count of 2 existing elements. If their count falls to 0, obviously its not a part of 2 elements anymore.
* The new element is *same as* one of the 2 elements. Increment the count of that element.

Consequently, the answer will be one of the 2 elements left behind. If they are not the answer, then there is no element with count > N / 3.

class Solution {

public:

struct eleCount {

int e; // Element

int c; // Count

}

int repeatedNumber(const vector<int> &V) {

if (V.size() == 0) return -1;

/\* Step 1: Create a temporary array (contains element

and count) of size 2. Initialize count of all

elements as 0 \*/

struct eleCount temp[2];

temp[0].c = temp[1].c = 0;

/\* Step 2: Process all elements of input array \*/

for (int i = 0; i < V.size(); i++) {

int j;

/\* If arr[i] is already present in

the element count array, then increment its count \*/

if (temp[0].e == arr[i] || temp[1] == arr[i]) {

if (temp[0].e == arr[i]) temp[0].c += 1;

else temp[1].c += 1;

} else {

/\* If arr[i] is not present in temp[] \*/

int l;

/\* If there is position available in temp[], then place

arr[i] in the first available position and set count as 1\*/

if (temp[0].c == 0 || temp[1].c == 0) {

int index = (temp[0].c == 0) ? 0 : 1;

temp[index].e = arr[i];

temp[index].c = 1;

} else {

/\* If all the position in the temp[] are filled, then

decrease count of every element by 1 \*/

temp[0].c -= 1;

temp[1].c -= 1;

}

}

}

/\*Step 3: Check actual counts of potential candidates in temp[]\*/

for (int i = 0; i < 2; i++) {

// Calculate actual count of elements

int ac = 0; // actual count

for (int j = 0; j < n; j++)

if (arr[j] == temp[i].e)

ac++;

// If actual count is more than n/k, then print it

if (ac > V.size() / 3) return temp[i].e;

}

// Not found

return -1;

}

};

**Closest Sum**

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target.  
Return the sum of the three integers.

*Assume that there will only be one solution*

**Example:**  
given array S = {-1 2 1 -4},  
and target = 1.

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2)

**Solution Hint**

The naive approach obviously is exloring all combinations of 3 integers using 3 loops.

Now, to look into improving, does it help if we sort the array?

**Solution Approach**

As stated in the earlier hint, the naive approach is to have 3 loops of i,j,k over the array. We then just track S[i]+S[j]+S[k] for the case when (S[i]+S[j]+S[k]-target) is minimum.  
The code for the same looks something like the following :

IF number of elements in S < 3

THEN return -1; // Invalid case

minDifference = abs(S[0] + S[1] + S[2] - target);

bestTillNow = S[0] + S[1] + S[2];

FOR i = 0 to size of S

FOR j = i + 1 to size of S

FOR k = j + 1 to size of S

newDiff = abs(S[i] + S[j] + S[k] - target)

IF newDiff < minDifference

minDifference = newDiff

bestTillNow = S[i] + S[j] + S[k]

END IF

END FOR

END FOR

END FOR

bestTillNow is my answer.

However, as stated earlier this approach is O(N^3). Lets see if we can do better.

Lets sort the array.  
When the array is sorted, try to fix the least integer by looping over it.  
Lets say the least integer in the solution is arr[i].

Now we need to find a pair of integers j and k, such that arr[j] + arr[k] is closest to (target - arr[i]).  
To do that, let us try the 2 pointer approach.  
If we fix the two pointers at the end ( that is, i+1 and end of array ), we look at the sum.

* If the sum is smaller than the sum we need to get to, we increase the first pointer.
* If the sum is bigger, we decrease the end pointer to reduce the sum.

**Solution Code**

class Solution {

public: int threeSumClosest(vector < int > & num, int target) {

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

int bestSum = 1000000000, sum = 0; // Fix the smallest number in the three integers

for (int i = 0; i < num.size() - 2; i++) { // Now num[i] is the smallest number in the three integers in the solution

int ptr1 = i + 1, ptr2 = num.size() - 1;

while (ptr1 < ptr2) {

sum = num[i] + num[ptr1] + num[ptr2];

if (abs(target - sum) < abs(target - bestSum)) {

bestSum = sum;

}

if (sum > target) {

ptr2--;

} else {

ptr1++;

}

}

}

return bestSum;

}

}

**Max Product Subarray[Dynamic Programming]**

Find the contiguous subarray within an array (containing at least one number) which has the largest product.  
Return an integer corresponding to the maximum product possible.

**Example :**

Input : [2, 3, -2, 4]

Return : 6

Possible with [2, 3]

**Solution Hint**

This problem can be solved in different ways:

1. DP based solution: Try to compute the maximum positive/negative product ending at any index i.
2. Observation based solution: Maintain something similar to slider and try to figure out when to change the position of slider. Keep maintaing positive/negative maximum product too.

**Solution Approach**

If there were no zeros or negative numbers, then the answer would definitely be the product of the whole array.

Now lets assume there were no negative numbers and just positive numbers and 0. In that case we could maintain a current maximum product which would be reset to A[i] when 0s were encountered.  
When the negative numbers are introduced, the situation changes ever so slightly. We need to now maintain the maximum product in positive and maximum product in negative. On encountering a negative number, the maximum product in negative can quickly come into picture.

**Solution Code**

class Solution {

public:

int maxProduct(int A[], int n) {

// store the result that is the max we have found so far

int r = A[0];

// imax/imin stores the max/min product of

// subarray that ends with the current number A[i]

for (int i = 1, imax = r, imin = r; i < n; i++) {

// multiplied by a negative makes big number smaller, small number bigger

// so we redefine the extremums by swapping them

if (A[i] < 0)

swap(imax, imin);

// max/min product for the current number is either the current number itself

// or the max/min by the previous number times the current one

imax = max(A[i], imax \* A[i]);

imin = min(A[i], imin \* A[i]);

// the newly computed max value is a candidate for our global result

r = max(r, imax);

}

return r;

}

};

# Technical Interview Questions

## Sub Trees

Question: Write a algorithm to determine if a tree is a sub-tree of another tree.

**public** **static** boolean containsTree(Node t1, Node t2) {  
    **if**(t2 == **null**)  
        **return** **true**;  
    **if**(t1 == **null**)  
        **return** **false**;  
    **if**(t1.data == t2.data)  
        **if**(matchTree(t1, t2))  
            **return** **true**;  
    **return** containsTree(t1.left, t2) || containsTree(t1.right, t2);  
}  
  
**private** **static** boolean matchTree(Node t1, Node t2) {  
    **if** (t1 == **null** && t2 == **null**)  
        **return** **true**;  
    **if** (t1 == **null** || t2 == **null**)  
        **return** **false**;  
    **if** (t1.data != t2.data)  
        **return** **false**;  
    **return** (matchTree(t1.left, t2.left) && matchTree(t1.right, t2.right));  
}

## ASCII

Question: Write functions to convert from ASCII to integer & integer to ASCII

**public** **static** int asciiToInt(String str) {  
    int result = 0;  
  
    **for**(int i = 0; i < str.length();i++) {  
        result\*= 10;  
        result+= str.charAt(i)-'0';  
    }  
  
    **return** result;  
}  
  
**public** **static** String intToAscii(int i) {  
    StringBuilder sb = **new** StringBuilder();  
  
    **while**(i != 0) {  
        char c = (char) ('0'+(i%10));  
        sb.insert(0,c);  
        i/= 10;  
    }  
  
    **return** sb.toString();  
}

## Reverse Linked List

Question: Write a function to reverse a singly-linked list.

**public** **static** ListNode reverseList(ListNode head) {  
        ListNode newHead = **null**;  
  
        **while**(head != **null**) {  
            ListNode temp = head.next;  
            head.next = newHead;  
            newHead = head;  
            head = temp;  
        }  
  
        **return** newHead;  
    }

## Reverse String

Write a function that takes a string as input and returns the string reversed.

Example: Given s = "hello", return "olleh".

Solution

public static String reverseString(String s) {

char[] array = s.toCharArray();

for(int i = 0; i < array.length/2; i++) {

char beginning = array[i];

char end = array[array.length - 1 - i];

array[i] = end;

array[array.length - 1 - i] = beginning;

}

return new String(array);

}

## Hamming Distance

The Hamming distance between two integers is the number of positions at which the corresponding bits are different.

Given two integers x and y, calculate the Hamming distance.

Note: 0 ≤ x, y < 2^31.

Example:

Input: x = 1, y = 4

Output: 2

Explanation:

1 (0 0 0 1)

4 (0 1 0 0)

↑ ↑

The above arrows point to positions where the corresponding bits are different.

Solution

public static int hammingDistance(int x, int y) {

int xor = x ^ y;

int count = 0;

while (xor != 0) {

if ((xor & 1) == 1) {

count ++;

}

xor = xor >> 1;

}

return count;

}

Solution 2

public static int hammingDistance(int x, int y) {

return (int) Integer.toBinaryString(x ^ y).chars().filter(e -> e == 49).count();

}

## Add Digits

Given a non-negative integer num, repeatedly add all its digits until the result has only one digit.

For example:

Given num = 38, the process is like: 3 + 8 = 11, 1 + 1 = 2. Since 2 has only one digit, return it.

Solution

public class Solution {

public int addDigits(int num) {

int x = 0;

while(num != 0) {

x+= (num % 10);

num /= 10;

if(num == 0 && x >= 10) {

num = x;

x = 0;

}

}

return x;

}

}

## Max Consecutive Ones

Given a binary array, find the maximum number of consecutive 1s in this array.

Example: Input: [1,1,0,1,1,1] Output: 3 Explanation: The first two digits or the last three digits are consecutive 1s. The maximum number of consecutive 1s is 3.

Note: The input array will only contain 0 and 1. The length of input array is a positive integer and will not exceed 10,000

Solution

public class Solution {

public int findMaxConsecutiveOnes(int[] nums) {

int max = 0;

int currentMax = 0;

for (int i : nums) {

if (i == 1) {

currentMax++;

if(currentMax > max) {

max = currentMax;

}

} else {

currentMax = 0;

}

}

return max;

}

}

## Two Sum

Given an array of integers, return indices of the two numbers such that they add up to a specific target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

Example: Given nums = [2, 7, 11, 15], target = 9, Because nums[0] + nums[1] = 2 + 7 = 9, return [0, 1].

Solution

class Solution {

public int[] twoSum(int[] nums, int target) {

Map<Integer, Integer> map = new HashMap<Integer, Integer>();

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

Integer k = map.get(target - nums[i]);

if(k != null) {

return new int[] {k, i};

}

map.put(nums[i], new Integer(i));

}

return null;

}

}

## Converging Linked Lists

Question: Write a function to determine whether two singly-linked lists converge. Write a function to find the intersection between two singly-linked lists.

**public** **class** **Node** {  
    int value;  
    Node next;  
}  
   
**public** **static** boolean listsConverge(Node head1, Node head2) {  
    **if**(head1 == **null** || head2 == **null**)  
        **return** **false**;  
   
    **while**(head1.next != **null**)  
        head1 = head1.next;  
   
    **while**(head2.next != **null**)  
        head2 = head2.next;  
   
    **return** head1.equals(head2);  
}  
   
**public** **static** Node listsIntersection(Node head1, Node head2) {  
    **if**(head1 == **null** || head2 == **null**)  
        **return** **null**;  
   
    int list1Length = 0;  
    int list2Length = 0;  
   
    Node n = head1;  
   
    **while**(n != **null**) {  
        list1Length++;  
        n = n.next;  
    }  
   
    n = head2;  
   
    **while**(n != **null**) {  
        list2Length++;  
        n = n.next;  
    }  
   
    **while**(list1Length > list2Length) {  
        head1 = head1.next;  
        list1Length--;  
    }  
   
    **while**(list2Length > list1Length) {  
        head2 = head2.next;  
        list2Length--;  
    }  
   
    **while**(head1 != **null**) {  
        **if**(head1.equals(head2))  
            **return** head1;  
   
        head1 = head1.next;  
        head2 = head2.next;  
    }  
   
    **return** **null**;  
}

## Longest Unique Substring

Given a string, find the length of the longest substring without repeating characters.

Examples:

Given "abcabcbb", the answer is "abc", which the length is 3.

Given "bbbbb", the answer is "b", with the length of 1.

Given "pwwkew", the answer is "wke", with the length of 3. Note that the answer must be a substring, "pwke" is a subsequence and not a substring.

class Solution {

public int lengthOfLongestSubstring(String s) {

int max = 0;

Set<Character> set = new HashSet<Character>();

int start = 0;

int end = 0;

while(end < s.length()) {

Character c = s.charAt(end);

if(set.contains(c)) {

while(set.contains(c)){

set.remove(s.charAt(start++));

}

}

set.add(new Character(c));

int len = end - start + 1; // Or equivalently, len = set.size();

if(max < len) {

max = len;

}

end++;

}

return max;

}

}

public static String findLongestUniqueSubstring(String str) {

java.util.Set<Character> set = new java.util.HashSet<Character>();

int currentStart = 0;

int length = 0;

int start = 0;

for(int currentEnd = 0; currentEnd < str.length(); currentEnd++) {

char c = str.charAt(currentEnd);

if(set.contains(c)) {

while(set.contains(c)) {

set.remove(str.charAt(currentStart++));

}

}

set.add(c);

if(currentEnd - currentStart + 1 > length) {

length = currentEnd - currentStart + 1;

start = currentStart;

}

}

return str.substring(start, start+length);

}

## Power Function

Question: Write a function to compute a^b efficiently. (A and B are both positive)

**public** **static** int power(int a, int b) {  
    **if**(b == 0)  
        **return** 1;  
   
    **if**(b == 1)  
        **return** a;  
   
    int result = power(a, b/2);  
   
    **if**(b % 2 == 0)  
        **return** result \* result;  
   
    **return** result \* result \* a;       
}

## Making Change

Question: For US currency, there are six coins that are in use today: 1¢, 5¢, 10¢, 25¢, 50¢, 1$. Write a function that returns the number of possible combinations of change that can be made from n ¢.

Example:

input: 7 output: 2 (7p, 1n 2p)

input: 10 output: 4 (1d, 10p, 2n, 1n 5p)

input: 15 output: 6 (1d 1n, 1d 5p, 3n, 2n 5p, 1n 10p, 15p)

**public** **static** int combinations(int n) {  
    **return** combinations(n,100);  
}  
   
**public** **static** int combinations(int n,int m) {  
    **if**(n < 0)  
        **return** 0;  
   
    **if**(n == 0)  
        **return** 1;  
   
    int combinations = 0;  
   
    **if**(m == 100)  
        combinations+= combinations(n-100,100);  
    **if**(m >= 50)  
        combinations+= combinations(n-50,50);  
    **if**(m >= 25)  
        combinations+= combinations(n-25,25);  
    **if**(m >= 10)  
        combinations+= combinations(n-10,10);  
    **if**(m >= 5)  
        combinations+= combinations(n-5,5);  
   
    combinations+= combinations(n-1,1);  
   
    **return** combinations;  
}

## Container With Most Water

Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of line i is at (i, ai) and (i, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container and n is at least 2.

public int maxArea(int[] height) {

int max = 0;

int i = 0;

int j = height.length - 1;

while (i < j) {

int k = Math.min(height[i], height[j]) \* (j - i);

if(k > max) {

max = k;

}

if(height[i]>=height[j]){

j--;

} else {

i++;

}

}

return max;

}

## Maximum Sub Array

Question: Given an array with at least one positive integer, find the contiguous sub-array with the largest sum. For example, for the sequence of values −2, 1, −3, 4, −1, 2, 1, −5, 4; the contiguous sub-array with the largest sum is 4, −1, 2, 1, with sum 6.

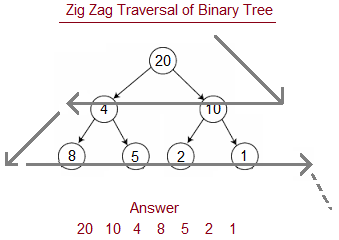
**public** **static** int [] maxSubArray (int [] array) {  
    int starting\_index = 0;  
    int length = 0;  
    int sum = 0;  
   
    int t\_starting\_index = 0;  
    int t\_length = 0;  
    int t\_sum = 0;  
   
    **for** (int i = 0; i < array.length; i++) {  
        **if**(t\_sum + array[i] > 0) {  
            t\_sum += array[i];  
            t\_length++;  
   
            **if**(t\_sum > sum) {  
                sum = t\_sum;  
                length = t\_length;  
                starting\_index = t\_starting\_index;  
            }  
        } **else** {  
            t\_sum = 0;  
            t\_length = 0;  
            t\_starting\_index = i + 1;  
        }  
    }  
   
    int [] sub = **new** int[length];  
   
    System.arraycopy(array, starting\_index, sub, 0, length);  
   
    **return** sub;  
}

Solution to find the max sum:

**public** **static** int maxSubArray(int [] array) {  
    int max\_ending\_here = 0;  
    int max\_so\_far = 0;  
   
    **for**(int i: array) {  
        max\_ending\_here = Math.max(0, max\_ending\_here + i);  
        max\_so\_far = Math.max(max\_so\_far, max\_ending\_here);  
    }  
   
    **return** max\_so\_far;  
}

## Zig-Zag Tree Traversal

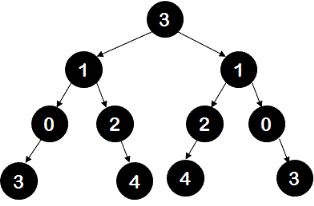
Question: Print out items of a Binary Tree in a zig-zag fashion.



**public** **static** void zigZag (Node root) {  
    Stack<Node> s1 = **new** Stack<Node>();  
    Stack<Node> s2 = **new** Stack<Node>();  
   
    s1.push(root);  
   
    **while**(!s1.isEmpty() || !s2.isEmpty()) {  
        **while**(!s1.isEmpty()) {  
            Node n = s1.pop();  
   
            System.out.println(n.data);  
   
            **if**(n.right != **null**)  
                s2.push(n.right);  
   
            **if**(n.left != **null**)  
                s2.push(n.left);  
        }  
   
        **while**(!s2.isEmpty()) {  
            Node n = s2.pop();  
   
            System.out.println(n.data);  
   
            **if**(n.left != **null**)  
                s1.push(n.left);  
   
            **if**(n.right != **null**)  
                s1.push(n.right);  
        }  
    }  
}

## Symmetric Tree

Write a program to check if the given binary tree is symmetric tree or not. A symmetric tree is defined as a tree which is mirror image of itself about the root node. For example, following tree is a symmetric tree.



**public** **static** boolean isMirror(Node n1, Node n2) {  
        **if**(n1 == **null** && n2 == **null**)  
            **return** **true**;  
   
        **if**(n1 == **null** || n2 == **null**)  
            **return** **false**;  
   
        **if**(n1.data != n2.data)  
            **return** **false**;  
   
        **return** isMirror(n1.left,n2.right) && isMirror(n1.right, n2.left);  
    }

## Stairs

Question: A man is walking up a set of stairs. He can either take 1 or 2 steps at a time. Given n number of stairs, find out how many combinations of steps he can take to reach the top of the stairs.

**public** **static** int combinations(int stairs) {  
    **if**(stairs == 0)  
        **return** 0;  
   
    int i = 0;  
    int j = 1;  
   
    **for**(int k = 0; k <= stairs; k++) {  
        int temp = j;  
        j+= i;  
        i = temp;  
    }  
    **return** i;  
}

## Curly braces

Question: Curly braces can be used in programming to provide scope-limit. Write a function to print all valid( properly opened and closed) combinations of n-pairs of curly braces.

Example:

input: 1 output: {}

input: 2 output: {}{}, Legacy Wiki template overrides

input: 3 output: {}{}{}, {}Legacy Wiki template overrides, Legacy Wiki template overrides{}, [[:Template:}{]], {{{}}}

input: 4 output: {}{}{}{}, {}{}Legacy Wiki template overrides, {}Legacy Wiki template overrides{}, {}[[:Template:}{]], {}{{{}}}, Legacy Wiki template overrides{}{}, Legacy Wiki template overridesLegacy Wiki template overrides, [[:Template:}{]]{}, [[:Template:}{}]], [[:Template:}Legacy Wiki template overrides]] {{{}}}{}, {Legacy Wiki template overrides{}}, {[[:Template:}{}]], [Template:Legacy Wiki template overrides](https://w.amazon.com/bin/view/Template:Legacy_Wiki_template_overrides)

**public** **static** void printBraces(int n) {  
        printBraces(**new** String(), n, n);  
    }  
   
    **public** **static** void printBraces(String str, int start, int end) {  
        **if** (start == 0 && end == 0) {  
            System.out.println(str);  
            **return**;  
        }  
   
        **if** (start > 0) {  
            printBraces(str+"{", start-1, end);  
        }  
   
        **if** (start < end) {  
            printBraces(str+"}", start, end-1);  
        }  
    }

## BST to Array and back

Question: Write a function to convert a sorted array to a BST of minimum height. Write a function to convert a BST to a sorted array.

**public** **static** Node arrayToBST(int [] array, int start, int end) {  
    **if**(end < start)  
        **return** **null**;  
   
    int mid = (start + end)/2;  
   
    Node n = **new** Node(array[mid]);  
   
    n.left = arrayToBST(array, start, mid-1);  
    n.right= arrayToBST(array, mid+1, end);  
   
    **return** n;  
}  
   
**public** **static** Integer [] BSTtoArray(Node root) {  
    Stack<Node> stack = **new** Stack<Node>();  
    Node node = root;  
   
    ArrayList<Integer> list = **new** ArrayList<Integer>();  
   
    **while**(!stack.isEmpty() || node != **null**) {  
        **if**(node != **null**) {  
            stack.push(node);  
            node = node.left;  
        } **else** {  
            node = stack.pop();  
            list.add(node.data);  
            node = node.right;  
        }  
    }  
   
    Integer [] array = **new** Integer [list.size()];  
    list.toArray(array);  
    **return** array;  
}

## Island Perimeter

You are given a map in form of a two-dimensional integer grid where 1 represents land and 0 represents water. Grid cells are connected horizontally/vertically (not diagonally). The grid is completely surrounded by water, and there is exactly one island (i.e., one or more connected land cells). The island doesn't have "lakes" (water inside that isn't connected to the water around the island). One cell is a square with side length 1. The grid is rectangular, width and height don't exceed 100. Determine the perimeter of the island.

Example

[0,1,0,0], [1,1,1,0], [0,1,0,0], [1,1,0,0](https://w.amazon.com/bin/view/0%2C1%2C0%2C0%5D%2C_%5B1%2C1%2C1%2C0%5D%2C_%5B0%2C1%2C0%2C0%5D%2C_%5B1%2C1%2C0%2C0)

Answer: 16 Explanation: The perimeter is the 16 yellow stripes in the image below:



**public** **static** int islandPerimeter(int[][] grid) {  
        int islandPerimeter = 0;  
        **for** (int i = 0; i < grid.length; i++) {  
            **for** (int j = 0; j < grid[i].length; j++) {  
                **if**(grid[i][j] == 1){  
                    **if** (i - 1 < 0 || grid[i-1][j] != 1) {  
                        islandPerimeter++;  
                    }  
                    **if** (j - 1 < 0 || grid[i][j-1] != 1) {  
                        islandPerimeter++;  
                    }  
                    **if** (j + 1 >= grid[i].length || grid[i][j+1] != 1 ){  
                        islandPerimeter++;  
                    }  
                    **if** (i + 1 >= grid.length || grid[i+1][j] != 1){  
                        islandPerimeter++;  
                    }  
                }  
            }  
        }  
        **return** islandPerimeter;  
    }

## Candy

There are N children standing in a line. Each child is assigned a rating value.

You are giving candies to these children subjected to the following requirements:

Each child must have at least one candy. Children with a higher rating get more candies than their neighbors. What is the minimum candies you must give?

Solution

**public** **static** int candy(int[] ratings) {  
    int[] left = **new** int[ratings.length];  
    int[] right = **new** int[ratings.length];  
      
    Arrays.fill(left, 1);  
    Arrays.fill(right, 1);  
      
    **for** (int i = 1; i < ratings.length; i++) {  
        **if** (ratings[i] > ratings[i-1]) {  
            left[i] = left[i-1] + 1;  
        }  
    }  
      
    **for** (int i = ratings.length -2; i >= 0; i--) {  
        **if** (ratings[i] > ratings[i+1]) {  
            right[i] = right[i+1] + 1;  
        }  
    }  
      
    int candies = 0;  
      
    **for** (int i = 0; i < ratings.length; i++) {  
        candies += Math.max(left[i], right[i]);  
    }  
    **return** candies;  
}

## Binary Search Tree

Question: Write a function to determine if a Binary Tree is a Binary Search Tree.

Solution

**class** **Node** {  
  int data;  
  Node left;  
  Node right;  
}  
   
**public** **static** boolean isBST(Node root, Integer min, Integer max) {  
    **if** (root == **null**)  
        **return** **true**;  
  
    **if**(min != **null** && min > root.data)  
        **return** **false**;  
  
    **if**(max != **null** && max <= root.data)  
        **return** **false**;  
  
    **return** isBST(root.left, min, root.data) && isBST(root.right, root.data, max);  
    }

## Hash Table

Write a basic implementation of HashTable

**public** **class** **HashTable** {  
    **private** **class** **Node** {  
        **public** Object key;  
        **public** Object value;  
        **public** Node next;  
   
        **public** Node (Object key, Object value) {  
            **this**.key = key;  
            **this**.value = value;  
        }  
    }  
   
    **private** Node[] buckets;  
    **private** int tableSize;  
    **private** int size;  
   
    **public** HashTable(int tableSize) {  
        **this**.tableSize = tableSize;  
        buckets = **new** Node[tableSize];  
        size = 0;  
    }  
   
    **public** void put(Object key, Object value) **throws** NullPointerException {  
        **if** (key == **null**)  
            **throw** **new** NullPointerException("Key cannot be null!");  
   
        int location = key.hashCode() % tableSize;  
   
        **if** (buckets[location] == **null**) {  
            buckets[location] = **new** Node(key, value);  
        } **else** {  
            Node n = buckets[location];  
            Node p = **null**;  
   
            **while** (n != **null**) {  
                **if** (n.key.equals(key)) {  
                    n.value = value;  
                    **return**;  
                }  
   
                p = n;  
                n = n.next;  
            }  
   
            p.next = **new** Node(key, value);  
        }  
   
        size++;  
    }  
   
    **public** Object get(Object key) **throws** NullPointerException {  
        **if** (key == **null**)  
            **throw** **new** NullPointerException("Key cannot be null!");  
   
        int location = key.hashCode() % tableSize;  
   
        Node n = buckets[location];  
   
        **while** (n != **null** && !n.key.equals(key))  
            n = n.next;  
   
        **if** (n == **null**)  
            **return** **null**;  
        **else**  
            **return** n.value;  
    }  
   
    **public** Object remove(Object key) **throws** NullPointerException {  
        **if** (key == **null**)  
            **throw** **new** NullPointerException("Key cannot be null!");  
   
        int location = key.hashCode() % tableSize;  
   
        Node n = buckets[location];  
        Node p = **null**;  
   
        **while** (n != **null** && !n.key.equals(key)) {  
            p = n;  
            n = n.next;  
        }  
   
        **if** (n == **null**)  
            **return** **null**;  
   
        size--;  
   
        **if** (p == **null**)  
            buckets[location] = n.next;  
        **else**  
            p.next = n.next;  
   
        **return** n.value;  
    }  
   
    **public** int size() {  
        **return** size;  
    }  
   
    **public** boolean isEmpty() {  
        **return** size == 0;  
    }

## Min Stack

Question: Write an implementation of a Stack that supports push(), pop(), and min() functions in constant time.

**public** **class** **Stack** {  
    **public** **class** **Node** {  
        Integer value;  
        Node next;  
  
        **public** Node(Integer value) {  
            **this**.value = value;  
        }  
    }  
  
    **private** Node head;  
    **private** Node minHead;  
  
    **public** Stack() {  
        head = **null**;  
        minHead = **null**;  
    }  
  
    **public** void push(Integer value) {  
        Node node = **new** Node(value);  
  
        **if**(head == **null**) {  
            head = node;  
            minHead = **new** Node(value);  
        } **else** {  
            node.next = head;  
            head = node;  
  
            **if** (value <= minHead.value) {  
                Node newMin = **new** Node(value);  
                newMin.next = minHead;  
                minHead = newMin;  
            }  
        }  
    }  
  
    **public** Integer pop() {  
        **if** (head == **null**)  
            **return** **null**;  
  
        Node n = head;  
        head = head.next;  
  
        **if** (n.value == minHead.value)  
            minHead = minHead.next;  
  
        **return** n.value;  
    }  
  
    **public** Integer min() {  
        **if** (minHead == **null**)  
            **return** **null**;  
  
        **return** minHead.value;             
    }  
}

## Unique Characters

Question: Implement an algorithm to determine if a string has all unique characters.

**public** **static** boolean isUnique(String str) {  
        boolean [] character\_set = **new** boolean[256];  
   
        **for**(int i = 0; i < str.length(); i++) {  
            int val = str.charAt(i);  
   
            **if**(character\_set[val])  
                **return** **false**;  
   
            character\_set[val] = **true**;  
        }  
        **return** **true**;  
    }

## Anagrams

Question: Write an algorithm to determine whether two strings are anagrams or not.

**public** **static** boolean isAnagram(String str1, String str2) {  
        **if** (str1.length() != str2.length())  
            **return** **false**;  
   
        int [] character\_set = **new** int[256];  
   
        **for**(int i = 0; i < str1.length(); i++) {  
            character\_set[str1.charAt(i)]++;  
            character\_set[str2.charAt(i)]--;  
        }  
   
        **for**(int i = 0; i < str1.length(); i++)  
            **if**(character\_set[str1.charAt(i)] != 0)  
                **return** **false**;  
   
        **return** **true**;  
    }

## Merge two sorted lists

Question: Write a function, that given two sorted lists of integers as input, returns a single sorted list with items from both lists with no duplicate elements.

Example: input: a = {1,2,3}; b = {4,5,6}; output: c = {1,2,3,4,5,6}; input: a = {7,8,9}; b = {1,8,20,24}; output: c = {1,7,8,9,20,24}; input: a = {3,3,4}; b = {4}; output: c = {3,4}; input: a = {1,2,2,3,3,4,5,6,7}; b = {4,5,6,7,8,8,8}; output: c = {1,2,3,4,5,6,7,8};

**public** **static** List<Integer> merge(int [] a, int [] b) {  
        List<Integer> list = **new** ArrayList<Integer>();  
   
        int a\_index = 0;  
        int b\_index = 0;  
   
        Integer i = **null**;  
   
        **while**(a\_index < a.length && b\_index < b.length) {  
            **if**(a[a\_index] < b[b\_index]) {  
                **if**(i == **null** || i < a[a\_index]) {  
                    i = a[a\_index];  
                    list.add(i);  
                }  
   
                a\_index++;  
            } **else** {  
                **if**(i == **null** || i < b[b\_index]) {  
                    i = b[b\_index];  
                    list.add(i);  
                }  
   
                b\_index++;  
            }  
        }  
   
        **while** (a\_index < a.length) {  
            **if** (i == **null** || i < a[a\_index]) {  
                i = a[a\_index];  
                list.add(i);  
            }  
            a\_index++;  
        }  
   
        **while** (b\_index < b.length) {  
            **if** (i == **null** || i < b[b\_index]) {  
                i = b[b\_index];  
                list.add(i);  
            }  
   
            b\_index++;  
        }  
   
        **return** list;  
    }

## Subsets

Find all subsets of a set

Example:

input: {a,b}

output: {}{a}{b}{a,b}

input: {a,b,c}

output: {}{a}{b}{c}{a,b}{a,c}{b,c}{a,b,c}

input:{abcd}

output:{} {d} {b} {c} {a} {bc} {da} {ca} {dc} {db} {ba} {dbca} {bca} {dba} {dca} {dbc}

**public** **static** Set<Set<Object>> getSubsets(Set<Object> set) {  
        Set<Set<Object>> subsets = **new** HashSet<Set<Object>>();   
   
        subsets.add(**new** HashSet<Object>());  
   
        **for**(Object o : set) {  
            Set<Set<Object>> temp = **new** HashSet<Set<Object>>();  
            **for**(Set<Object> s: subsets)  
                temp.add(**new** HashSet<Object>(s));  
   
            **for** (Set<Object> s : temp)  
                s.add(o);  
   
            subsets.addAll(temp);  
        }         
        **return** subsets;  
    }  
      
    **public** **static** Set<Set<Object>> getSubsets(List<Object> list) {  
        Set<Set<Object>> subsets = **new** HashSet<Set<Object>>();  
   
        **for**(int i = 0; i < Math.pow(2, list.size()); i++) {  
            Set<Object> subset = **new** HashSet<Object>();  
   
            int j = 1;  
            **for**(int k = 0; k < list.size(); k++) {  
                **if**((j & i) != 0) {  
                    subset.add(list.get(k));  
                }  
                j = j << 1;  
            }  
            subsets.add(subset);  
        }  
        **return** subsets;  
    }

## String Permutations

Question: Find all permutations of a String.

Example:

input: abcd

output: dcba, cdba, cbda, cbad, dbca, bdca, bcda, bcad, dbac, bdac, badc, bacd, dcab, cdab, cadb, cabd, dacb, adcb, acdb, acbd, dabc, adbc, abdc, abcd

**public** **static** ArrayList<String> findPermutations(String str) {  
    ArrayList<String> permutations = **new** ArrayList<String>();  
    permutations.add(**new** String());  
  
    char [] strArray = str.toCharArray();  
  
    **for**(char c : strArray) {  
        ArrayList<String> tempList = **new** ArrayList<String>();  
  
        **for**(String s: permutations) {  
            **for**(int i = 0; i < s.length(); i++)  
                tempList.add(**new** String(s.substring(0, i) + c + s.substring(i, s.length())));  
            tempList.add(**new** String(s+c));  
        }  
        permutations = tempList;  
    }  
    **return** permutations;  
}

## Array Intersection

Question: Find the intersection of two unsorted integer arrays.

**public** **static** Set<Integer> arrayIntersection(int [] list1, int [] list2) {  
    Set<Integer> output = **new** HashSet<Integer>();  
  
    Set<Integer> set = **new** HashSet<Integer>();   
  
    **for**(int i : list1)  
        set.add(i);  
  
    **for**(int i: list2)  
        **if**(set.contains(i))  
            output.add(i);  
  
    **return** output;  
}

## Sorting

Question: Write an efficient sorting algorithm.

**public** **static** int[] quickSort(int [] array, int low, int high) {  
        **if**(array == **null**)  
            **return** **null**;  
   
        **if**(array.length==0)  
            **return** array;  
   
        int i = low, j = high;  
   
        int pivot = array[low + (high-low)/2];  
   
        **while** (i <= j) {  
            **while** (array[i] < pivot)  
                i++;  
   
            **while** (array[j] > pivot)  
                j--;   
   
            **if** (i <= j) {  
                int temp = array[i];  
                array[i] = array[j];  
                array[j] = temp;  
                i++;  
                j--;  
            }  
        }  
   
        **if** (low < j)  
            quickSort(array, low, j);  
   
        **if** (i < high)  
            quickSort(array, i, high);  
   
        **return** array;  
    }  
   
    **public** **static** int[] mergeSort(int [] array) {  
        **if**(array == **null**)  
            **return** **null**;  
   
        **if**(array.length < 2)  
            **return** array;  
   
        int [] array1 = **new** int[array.length/2];  
   
        System.arraycopy(array, 0, array1, 0, array1.length);  
   
        int [] array2;  
   
        **if**(array.length%2==0)  
            array2 = **new** int[array.length/2];  
        **else**   
            array2 = **new** int[array.length/2+1];  
   
        System.arraycopy(array, array1.length, array2, 0, array2.length);  
   
        array1 = mergeSort(array1);  
        array2 = mergeSort(array2);  
   
        int a = 0;  
        int a1 = 0;  
        int a2 = 0;  
   
        **while**(a1 < array1.length && a2 < array2.length)  
            **if**(array1[a1]<array2[a2])  
                array[a++]=array1[a1++];  
            **else**  
                array[a++]=array2[a2++];  
   
        **while**(a1 < array1.length)  
            array[a++]=array1[a1++];  
   
        **while**(a2 < array2.length)  
            array[a++]=array2[a2++];  
   
        **return** array;  
    }

## Missing Number

Question: Find the missing number in an array of consecutive integers.

Example:

input:2,3,4,5,6,7,8,9,10,11,12,13,14,15 output: null

input:2,3,5,6,7,8,9,10,11,12,13,14,15 output: 4

input:2,3,4,5,6,7,8,9,10,11,12,14,15 output: 13

**public** **static** Integer missingNumber(int [] array) {  
        int low = 0;  
        int high = array.length -1;  
   
        int mid;  
   
        **while**(low<high) {  
            mid = (low+high)/2;  
   
            **if**(mid-low == array[mid]- array[low]) {  
                **if**( mid+1< array.length && array[mid]+1 != array[mid+1])  
                    **return** array[mid]+1;  
                **else**  
                    low = mid+1;  
            } **else** {  
                **if**(mid -1 > -1 && array[mid]-1 != array[mid-1])  
                    **return** array[mid]-1;  
                **else**  
                    high = mid-1;  
            }  
        }  
   
        **return** **null**;  
    }

## Missing Spaces

Question: You are given a sentence with no spaces and dictionary containing thousands of words. Write an algorithm to reconstruct the sentence by inserting spaces in the appropriate positions.

Example:

input: "theskyisblue" output: "the sky is blue"

input: "thegrassisgreen" output: "the grass is green"

**public** **static** String makeSentence(String str, Set<String> dictionary) {  
        char [] array = str.toCharArray();  
   
        StringBuilder prefix = **new** StringBuilder();  
   
        **for**(int i = 0; i < array.length; i++) {  
            prefix.append(array[i]);  
   
            **if**(dictionary.contains(prefix.toString())) {  
                **if** (prefix.length() == array.length)  
                    **return** prefix.toString();  
   
                String suffix = makeSentence(**new** String(array,prefix.length(),array.length-prefix.length()), dictionary);  
   
                **if** (suffix != **null**) {  
                    prefix.append(" ");  
                    prefix.append(suffix);  
                    **return** prefix.toString();  
                }  
            }  
        }  
   
        **return** **null**;  
    }

## Nth smallest element in a binary search tree

Question: Write a function to find the nth smallest element in a binary search tree

**class** **Node** {  
    int i;  
    Node left;  
    Node right;  
}  
  
**public** **static** Node nthSmallestElement(Node root, int n) {  
    Stack<Node> stack = **new** Stack<Node>();  
    Node node = root;  
    int i = 0;  
   
    **while** (!stack.isEmpty() || node != **null**) {  
        **if** (node != **null**) {  
            stack.push(node);  
            node = node.left;  
        } **else** {  
            node = stack.pop();  
            **if**(++i == n) {  
                **return** node;  
            }  
            node = node.right;  
        }  
    }  
   
    **return** **null**;  
}

## Implement a Queue

**public** **class** **Node** {  
    **public** Node next;  
    **public** Object data;  
}  
  
**public** **class** **Queue** {  
    **private** Node start;  
    **private** Node end;  
      
    **public** Queue() {  
        start = **null**;  
        end = **null**;  
    }  
      
    **public** void add(Object o) {  
        Node newNode = **new** Node();  
        newNode.data = o;  
          
        **if** (start == **null**) {  
            start = newNode;  
            end = newNode;  
        } **else** {  
            end.next = newNode;  
            end = newNode;  
        }  
    }  
      
    **public** Object remove() {  
        **if** (start == **null**) {  
            **return** **null**;  
        }  
          
        Object o = first.data;  
        first = first.next;  
        **return** o;  
    }  
}

## LinkedList of Nodes in a Binary Tree

Given a binary tree, design an algorithm which creates a linked list of all the nodes at each depth.

**public** **static** ArrayList<LinkedList<Node>> makeLinkedLists(Node root) {  
    LinkedList<Node> current = **new** LinkedList<Node>();  
    LinkedList<Node> nextLevel = **new** LinkedList<Node>();  
    ArrayList<LinkedList<Node>> lists = **new** ArrayList<LinkedList<Node>>();  
  
    current.add(root);  
  
    **while** (!current.isEmpty()) {  
        lists.add(current);  
          
        **for** (Node n : current)) {  
            **if** (n.left != **null**) nextLevel.add(n.left);  
            **if** (n.right != **null**) nextLevel.add(n.right);  
        }  
  
        current = nextLevel;  
        nextLevel = **new** LinkedList<Node>();  
    }  
    **return** lists;  
}

## Inorder Successor in Binary Search Tree

Given a binary search tree and a node in it, find the in-order successor of that node in the BST.

**public** **static** Node inorderSuccessor(Node root, Node n) {  
    **if**(n.right != **null**){  
        n = n.right;  
        **while**(n.left !=**null**){  
            n = n.left;  
        }  
        **return** n;  
    }  
    Node successor = **null**;  
    **while**(root != **null**) {  
        **if**(n.data < root.data){  
            successor = root;  
            root = root.left;  
        } **else** **if**(n.data > root.data){  
            root = root.right;  
        } **else** {  
            **break**;  
        }  
    }  
    **return** successor;  
}

## Find min element in rotated, sorted array

Find the minimum element in a rotated, sorted array.

Example: 4,5,6,7,8,2,3 Answer: 2

**public** **static** int (int[] array) {  
    int start = 0;  
    int end = array.length - 1;  
  
    **while**(start < end) {  
        **if** (array[start] < array[end]) {  
            **return** array[start];  
        }  
          
        int mid = array[(start + end)/2];  
        **if** (array[mid] >= array[start]) {  
            start = mid+1;  
       } **else** {  
           end = mid;  
       }  
    }  
    **return** num[start];  
}

## Palindrome

Question: Write a function to find out whether a string is a palindrome.

**public** **static** boolean isPalindrome(String input) {  
        int i = 0;  
        int j = input.length() - 1;  
   
        **while** (i < j) {  
            **if** (input.charAt(i) != input.charAt(j)) {  
                **return** **false**;  
            }  
            i++;  
            j--;  
        }  
   
        **return** **true**;  
    }

## Heap

Implement a min heap

**public** **class** **BinaryHeap**<T **extends** Comparable<T>>{  
    **private** **static** **final** int DEFAULT\_CAPACITY = 10;  
    **private** int size;  
    **private** T[] heap;  
  
    **public** BinaryHeap() {  
        size = 0;  
        heap = (T[]) **new** Comparable[DEFAULT\_CAPACITY];  
    }  
  
    **public** T peek() {  
        **if** (**this**.isEmpty()) {  
            **return** **null**;  
        }  
        **return** heap[0];  
    }  
  
    **public** void add(T value) {  
        **if** (size >= heap.length) {  
            heap = **this**.resize();  
        }  
  
        heap[size++] = value;  
        bubbleUp();  
    }  
  
    **public** T remove() {  
        T result = peek();  
  
        heap[0] = heap[size - 1];  
        heap[size - 1] = **null**;  
        size--;  
  
        bubbleDown();  
  
        **return** result;  
    }  
  
    **public** boolean isEmpty() {  
        **return** size == 0;  
    }  
  
    **public** int size() {  
        **return** size;  
    }  
  
    **private** T[] resize() {  
        **return** java.util.Arrays.copyOf(heap, heap.length \* 2);  
    }  
  
    **private** boolean hasParent(int i) {  
        **return** i > 0;  
    }  
  
    **private** int leftIndex(int i) {  
        **return** (i \* 2) + 1;  
    }  
  
    **private** int rightIndex(int i) {  
        **return** (i \* 2) + 2;  
    }  
  
    **private** boolean hasLeftChild(int i) {  
        **return** leftIndex(i) < size;  
    }  
  
    **private** boolean hasRightChild(int i) {  
        **return** rightIndex(i) < size;  
    }  
  
    **private** T parent(int i) {  
        **return** heap[parentIndex(i)];  
    }  
  
    **private** int parentIndex(int i) {  
        **return** (i-1) / 2;  
    }  
  
    **private** void swap(int index1, int index2) {  
        T tmp = heap[index1];  
        heap[index1] = heap[index2];  
        heap[index2] = tmp;  
    }  
  
    **private** void bubbleUp() {  
        int index = **this**.size - 1;  
  
        **while** (hasParent(index)  
                && (parent(index).compareTo(heap[index]) > 0)) {  
            swap(index, parentIndex(index));  
            index = parentIndex(index);  
        }  
    }  
  
    **private** void bubbleDown() {  
        int index = 0;  
  
        **while** (hasLeftChild(index)) {  
            int smallerChild = leftIndex(index);  
  
            **if** (hasRightChild(index)  
                    && heap[leftIndex(index)].compareTo(heap[rightIndex(index)]) > 0) {  
                smallerChild = rightIndex(index);  
            }  
  
            **if** (heap[index].compareTo(heap[smallerChild]) > 0) {  
                swap(index, smallerChild);  
            } **else** {  
                **break**;  
            }  
  
            index = smallerChild;  
        }  
    }  
}

## Median Value

Question: Write a function to find the median value from a stream of integers.

**public** **static** Double median (Iterator<Integer> stream){  
    Queue<Integer> minHeap = **new** PriorityQueue<Integer>();  
    Queue<Integer> maxHeap = **new** PriorityQueue<Integer>(20, Collections.reverseOrder());  
  
    **while**(stream.hasNext()){  
        maxHeap.add(stream.next());  
  
        **if** (!minHeap.isEmpty() && maxHeap.peek() > minHeap.peek()){  
            Integer maxHeapRoot = maxHeap.poll();  
            Integer minHeapRoot = minHeap.poll();  
            maxHeap.add(minHeapRoot);  
            minHeap.add(maxHeapRoot);  
        }  
  
        **if**(maxHeap.size() - minHeap.size() > 1) {  
            minHeap.add(maxHeap.poll());  
        }  
    }  
  
    **if** (maxHeap.size() > minHeap.size()){  
        **return** maxHeap.peek().doubleValue();  
    }  
  
    **return** (maxHeap.peek() + minHeap.peek()) / 2.0;  
}

## Calculate Angle

Given the time, calculate the smaller angle between the hour and minute hands on an analog clock.

**public** **static** int (int hour, int min) {  
    int minuteAngle = min \* 6;  
    int hourAngle = hour \* 30 + (min / 2);  
      
    int angle = Math.abs(minuteAngle - hourAngle);  
    **return** Math.min(360 -  angle, angle);  
}

## Implement a LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

**class** **Node**{  
    int key;  
    int value;  
    Node pre;  
    Node next;  
   
    **public** Node(int key, int value){  
        **this**.key = key;  
        **this**.value = value;  
    }  
}  
  
**public** **class** **LRUCache** {  
    int capacity;  
    HashMap<Integer, Node> map = **new** HashMap<Integer, Node>();  
    Node head=**null**;  
    Node end=**null**;  
   
    **public** LRUCache(int capacity) {  
        **this**.capacity = capacity;  
    }  
   
    **public** int get(int key) {  
        **if**(map.containsKey(key)){  
            Node n = map.get(key);  
            remove(n);  
            setHead(n);  
            **return** n.value;  
        }  
   
        **return** -1;  
    }  
   
    **public** void remove(Node n){  
        **if**(n.pre!=**null**){  
            n.pre.next = n.next;  
        }**else**{  
            head = n.next;  
        }  
   
        **if**(n.next!=**null**){  
            n.next.pre = n.pre;  
        }**else**{  
            end = n.pre;  
        }  
   
    }  
   
    **public** void setHead(Node n){  
        n.next = head;  
        n.pre = **null**;  
   
        **if**(head!=**null**)  
            head.pre = n;  
   
        head = n;  
   
        **if**(end ==**null**)  
            end = head;  
    }  
   
    **public** void set(int key, int value) {  
        **if**(map.containsKey(key)){  
            Node old = map.get(key);  
            old.value = value;  
            remove(old);  
            setHead(old);  
        }**else**{  
            Node created = **new** Node(key, value);  
            **if**(map.size()>=capacity){  
                map.remove(end.key);  
                remove(end);  
                setHead(created);  
   
            }**else**{  
                setHead(created);  
            }      
   
            map.put(key, created);  
        }  
    }  
}

## Rotate Array

Write a function to rotate an array to the left n-times.

**static** int[] rotLeft(int[] array, int n) {  
        int[] rotatedArray = **new** int[array.length];  
        **for**(int index = 0; index < array.length; index++) {  
            int newPos = (index + (array.length - n)) % array.length;  
            rotatedArray[newPos] = array[index];  
        }  
        **return** rotatedArray;  
    }

## Rotate 2D Array

Write a function to rotate an NxN array 90 degrees.

**public** **static** void rotate(int[][] matrix, int n) {  
    **for** (int layer = 0; layer < n / 2; ++layer) {  
        int first = layer;  
        int last = n - 1 - layer;  
        **for**(int i = first; i < last; ++i) {  
            int offset = i - first;  
            int top = matrix[first][i]; // save top  
  
            // left -> top  
            matrix[first][i] = matrix[last-offset][first];            
            // bottom -> left  
            matrix[last-offset][first] = matrix[last][last - offset];   
  
            // right -> bottom  
            matrix[last][last - offset] = matrix[i][last];   
  
            // top -> right  
            matrix[i][last] = top; // right <- saved top  
        }  
    }  
}

## Implement a Queue using two stacks

**public** **class** **TwoStacks** {  
    Stack<Object> pushStack;  
    Stack<Object> popStack;  
  
    **public** TwoStacks (){  
        pushStack = **new** Stack<>();  
        popStack = **new** Stack<>();  
    }  
  
    **public** void push(Object o) {  
        pushStack.push(o);  
    }  
  
    **public** Object pop() {  
        **if** (!popStack.isEmpty()) {  
            **return** popStack.pop();  
        }  
        **while**(!pushStack.isEmpty()){  
            popStack.push(pushStack.pop());  
        }  
        **if** (!popStack.isEmpty()) {  
            **return** popStack.pop();  
        }  
        **return** **null**;  
    }  
}

## Balanced Binary Tree

Implement an algorithm to determine is a binary tree is balanced. A balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.

**public** **static** boolean isBalanced(Node root){  
    **return** (getHeight(root) != -1);  
}  
**public** **static** int getHeight(Node root) {  
    **if** (root == **null**) {  
        **return** 0;  
    }  
      
    int leftHeight = getHeight(root.left);  
    **if** (leftHeight == -1) {  
        **return** -1;  
    }  
      
    int rightHeight = getHeight(root.right);  
    **if** (rightHeight == -1) {  
        **return** -1;  
    }  
      
    **if** (Math.abs(leftHeight - rightHeight) > 1) {  
        **return** -1;  
    }  
      
    **return** Math.max(leftHeight, rightHeight) + 1;  
}

## Lowest Common Ancestor

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

**public** TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {  
    **if** (root == **null** || root == p || root == q) **return** root;  
    TreeNode left = lowestCommonAncestor(root.left, p, q);  
    TreeNode right = lowestCommonAncestor(root.right, p, q);  
    **return** left == **null** ? right : right == **null** ? left : root;  
}

## Fibonacci

Implement a function which returns the nth number in Fibonacci sequences with an input n.

**public** **static** int fibonacci(int n) {  
    **if** (n == 0 || n == 1) {  
        **return** n;  
    }  
  
    int fibMinus1 = 1;  
    int fibMinus2 = 0;  
  
    int fibN = 0;  
  
    **for** (int i = 2; i <= n; i++) {  
        fibN = fibMinus1 + fibMinus2;  
        fibMinus2 = fibMinus1;  
        fibMinus1 = fibN;  
    }  
  
    **return** fibN;  
}

## First Unique Character

Find the first unique character in a string.

**public** **static** Character firstUniqueCharacter(char[] string) {  
    Map<Character, Integer> map = **new** HashMap<Character, Integer>();  
    **for** (char c: string) {  
        Integer i = map.get(c);  
        map.put(c, (i==**null**) ? 1: 1+i);  
    }  
    **for** (char c: string) {  
        Integer i = map.get(c);  
        **if** (i == 1) {  
            **return** c;  
        }  
    }  
    **return** **null**;  
}

## Largest number in BST which is less than or equal to N

We have a binary search tree and a number N. Our task is to find the greatest number in the binary search tree that is less than or equal to N. Print the value of the element if it exists otherwise print -1.

**public** **static** int (Node root, int n) {  
    int val = -1;  
    **while**(root != **null**) {  
        **if**(root.data > n) {  
            root = root.left;  
        } **else** {  
            val = root.data;  
            root = root.right;  
        }  
    }  
    **return** val;  
}

**Merge sorted arrays**

**Question (Coding)**

You’re given two sorted arrays of integers, write a method that will merge these arrays together and return the resulting sorted array.

**Focus**

Optimisation, Coding mechanics

**Solution**

The *correct* solution is generally some flavour of putting a pointer at the head of each array and comparing the values at their indices repeatedly, storing the smaller value in a new array and advancing the pointer as you go. One array will run out before the other so there’s usually a need to have a dedicated piece of code with will append whatever is left in the non-empty array to the end of the newly constructed array. Not all solutions will require this step to be a separate bit of code, but most will.

Make sure that you ask what the overall runtime of the solution is. Candidates should be able to identify it as O(n).

**Common Mistakes**

* There are many ways of solving this problem, but only solutions that run in O(n) time should be given credit. For example, a solution that involves naively combining the arrays and then sorting them again should not be accepted. It’s ok to reiterate to the candidate that the arrays are pre-sorted and that some speedup beyond normal sorting techniques should be achieved, but if they can’t get to O(n) there’s no need to keep asking questions.
* Candidates often completely neglect keeping track of the array bounds. If you can give some simple hints to the candidate and they fix can their code, the solution is probably still acceptable. If after a hint they still don’t understand that this question is all about pointer management then it’s probably safe to say that they got this question wrong.
* Some candidates will complete the first step correctly, but forget to have the append step at the end. You can have them walk through an example. If they don’t pick up on their error after that hint then they’ve gotten the question wrong.

**Analysis**

This question is fairly straight forward so it’s imperative that you have the candidate actually code their solution. If a candidate can get to a working solution, even if they need a couple of hints, that’s fine and credit should be given. If they struggle and need their hand held throughout the question, then the candidate shouldn’t be given credit. Additionally, if it takes them an inordinate amount of time to get to a solution, say over 15 minutes then that should be taken in to consideration as well.

**Sample solution (Java)**

**public** **static** int[] merge(int[] a, int[] b) {  
  
    int[] answer = **new** int[a.length + b.length];  
    int i = 0, j = 0, k = 0;  
  
    **while** (i < a.length && j < b.length) {  
        **if** (a[i] < b[j]) {  
            answer[k++] = a[i++];  
        } **else** {  
            answer[k++] = b[j++];  
        }  
    }  
  
    **while** (i < a.length)    
        answer[k++] = a[i++];  
  
    **while** (j < b.length)      
        answer[k++] = b[j++];  
  
    **return** answer;  
}

Seeing that we are looking for optimal solutions, I would expect a good candidate to use [System.arraycopy](http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#arraycopy%28java.lang.Object,%20int,%20java.lang.Object,%20int,%20int%29) for handling the remaining elements to be copied. Further to special case empty arrays, eg. merge([],[1,2,3]) and non overlapping arrays, eg merge([7,8,9],[1,2,3]). This may (hopefully) lead to a discussion around need for a defensive copy of the input parameter (see [Array.copyOf](http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#copyOf%28int%91%93,%20int%29)).

**Reverse a doubly linked list**

**Question (Coding)**

Reverse a doubly linked list in place.

**Focus**

Coding mechanics, Pointers, linked list data structure

**Solution**

We're looking for an iterative solution here, the question is quite unambiguous. There aren’t a lot of creative ways of doing this, this is basically a right or wrong type of question.

* Have 3 points: previous, current, next.
* Set current to the first node.
* While current != null.
  1. Set temp to current.next
  2. Set current.next to current.previous
  3. Set current.previous to temp
  4. Set current to temp

*alternative (below) uses temp to hold onto current.previous*

**Common Mistakes**

* The most common mistakes involve not understanding the question. Thoroughly explaining what you’re looking for is ok. Encouraging candidates to write out examples is also a good approach. Some candidates try to do this with too few pointers and simply miss that their solution won’t work.
* Some candidates try to waive their hands at the problem without clearly articulating what they’re actually doing. It’s very important to have the candidates actually **write** the code for this question and read it back. Sometimes it can help to tell the candidate to write down an example and walk through it, but they really should be doing this on one their own.
* Returning current at the end of the function instead of temp->previous

**Analysis**

They either get this one right or wrong. You can always count the pointers to see if they did it right. If they don’t have 3 references to things then they didn’t answer the question correctly.

**Sample solution (Java)**

/\*   
 \* Function to reverse a Doubly Linked List   
 \*/  
**public** Node<T> reverse(Node<T> head) {  
    Node temp = **null**;  
    Node current = head;  
      
    /\* Swap next and prev for all nodes of doubly linked list \*/  
     **while** (current !=  **null**) {  
       temp = current->prev;  
       current->prev = current->next;  
       current->next = temp;  
       current = current->prev;  
     }        
          
    **if** (temp == **null** )   
        **return** head  
    **return** temp->prev;  
}

**Walkthrough**

**Input:**

* null <- 1 <-> 2 <-> 3 -> null

**End of each iteration:**

1. 2 <- 1 -> null
2. 3 <- 2 <-> 1 -> null
3. null <- 3 <-> 2 <-> 1 -> null *and current is* ***null*** *and temp is pointing to* ***2****.*

**Single linked listed reversal**

I prefer a single liknked list variation of this problem as it tests the candidates ability to reason about the ordering and logic. For fun here is a ruby solution to the problem

**class** **Node**  
  attr\_accessor :data, :next  
  
  **def** initialize(d, n=nil)   
    @data = d  
    @next = n  
  **end**  
  
  **def** to\_s  
    next\_s = " -> **#{**@next.to\_s**}**" **if** @next  
    data.to\_s + next\_s.to\_s  
  **end**  
**end**  
  
# in place reverse the linked list  
**def** reverse!(list)  
  **return** list **if** list.nil? || list.next.nil?  
     
   current = list.next  
   list.next = nil  
  
   **while** current **do**  
     new\_head = current  
     current = current.next  
     new\_head.next = list  
     list = new\_head  
   **end**  
   list  
**end**  
  
puts reverse!(nil)  
puts reverse!(Node.new(1))  
list = Node.new(1,Node.new(2,Node.new(3)))  
puts "List **#{**list**}** ... reversed: **#{**reverse!(list)**}**"

**Find the nth last element in a linked list**

**Question**

Write a function that finds the nth last element in a linked list. For example, given a list with elements foo -> bar -> quux -> baz and n = 2, the function should return the quux element.

**Focus**

Coding mechanics, Linked list data structures

**Solution**

We are looking for a solution for a singly linked list where we track both the current tail node and tail-n node.

If the candidate asks whether the list is a doubly-linked list or not, ask them why they're asking. They may be making an assumption that they have the tail of the list and that they can simply traverse backwards by 'n' elements. It's great if they ask the question, but not so great if they don't spell out their assumption. Insist that we don't have direct access to the tail, then ask them if it still matters whether the list is singly- or doubly-linked.

Weaker solutions include:

* Using a data structure (such as a stack) to capture elements during a forward traversal of the list (then popping n elements off the stack).
* counting the elements in the list and then traversing a second time to return length-n+1

Challenge the candidate to come up with something more efficient and to contrast the performance differences between the solutions.

**Common Mistakes**

* Not stating their assumptions and coding a non-optimal or incorrect solution.
* Being one off, get them to iterate through examples
* n<=0 should not be valid
* not coping with length<n

**Sample solution (Java)**

// @param n position from end, counting from 1  
**public** **static** Node<T> getNthLastElement(Node<T> headNode, int n) {  
 **if** (n <= 0) {  
 **return** **null**;  
 }  
   
 Node<T> lastNode = headNode;  
 Node<T> nthLast = lastNode;  
 **while** (lastNode != **null** && n > 0) {  
 lastNode = lastNode.getNext();  
 n--;  
 }  
  
 **if** (lastNode == **null** && n > 0) {  
 **return** **null**;  
 }  
   
 **while** (lastNode != **null**) {  
 lastNode = lastNode.getNext();  
 nthLast = nthLast.getNext();   
 }  
 **return** nthLast;  
}

**Convert an IP address String to an integer**

**Question**

Write some code to convert an IPv4, Octet style, address into its equivalent integer value

For E.g. "10.0.0.1" would result in an integer value of 167772161

**Focus**

Coding mechanics, Internet domain knowledge, Bits & Bytes, Text Parsing

**Solution**

We're looking for "sane" solutions for the given language and

* Valid text-parsing logic with checks for edge-cases.
* Bounds checking for loops.
* Should be able to describe the computational complexity big-O.
* Loops should have obvious termination cases.
* Efficient, Readable
* Test cases
  1. empty string
  2. too few sections, too many
  3. invalid characters in sections (bonus if they can easily handle 001 as invalid)
  4. out of bounds integers

**Sample solution (Java)**

**public** **class** **IpConversion** {  
    
  **private** **static** Pattern dotSepPattern = Pattern.compile("\\.");  
  
  **public** **static** int parseOctet(String s) **throws** NumberFormatException  {  
    int result = 0;  
    int max = s.length();  
  
    **if** (max > 0 && max < 4) {  
      int i = 0;  
        **while** (i < max) {  
          //int digit = Character.digit(s.charAt(i),10);  
          int digit = s.charAt(i) - '0';  
          result = result\*10 + digit;  
  
          // Tricky … handle 001 which should not be accepted, yet accept 0  
          **if** ((result == 0 && digit == 0 && i != max-1) ||  
               digit < 0 || digit > 9 || result > 255) {  
            **throw** **new** NumberFormatException(s);  
          }  
  
          i++;  
        }  
        **return** result;  
    }   
    **throw** **new** NumberFormatException(s);  
  }   
  
  **public** **static** int ipToInt(String addr) {  
    String[] addrOctets = dotSepPattern.split(addr, 0);  
      **if** (addrOctets.length == 4) {           
        int numericAddr = 0;          
        **for** (String octet: addrOctets) {  
          numericAddr = (numericAddr << 8) | (parseOctet(octet));  
        }  
        **return** numericAddr;  
    }  
    **throw** **new** IllegalArgumentException("Malformed IP address");  
  }  
}

**Balanced brackets**

**Question**

Write a function which determines whether a mathematical string/expression has balanced brackets

For example, given the input:

* 1+ 1)\*2 the function would return **false** (i.e. unbalanced)
* 1 + (3 + 3)\*2 would return **true** (i.e. balanced)

**Focus**

Coding mechanics

**Solution**

We're looking for a single parse of the input string and tracking the bracket count as we go.

* An extension to this is to determine how to handle and ignore braces in strings. It should be as simple as adding another case to the switch statement, one two handle string delimiter
* Even better would be how to handle escaping the string delimiter

but as we want to keep the question short, they can just talk through the enhancements.

**Common mistakes**

* not aborting as soon as we detect negative balance, (e.g. first example)

**Sample solution (Java)**

**public** boolean balance(**final** String s) {  
  int openBrackets = 0;  
  **for** (int i=0; i < s.length; i++) {  
    **switch** (s.charAt(i)) {  
      **case** '(':  
        openBrackets++;  
        **break**;  
      **case** ')':  
        **if** (--openBrackets < 0) {  
          **return** **false**;  
        }  
        **break**;  
    }  
  }  
  **return** (openBrackets == 0);  
}

**1 Count occurrences of letters in a string**

**1.1 Question**

Write a program that prints out the occurrences of letters in a string

For example, given the input Amazon Web Services, the program should output (excerpt):

Character 'a' appears 2 times

Character 'b' appears 1 time

Character 'c' appears 1 time

Character 'd' appears 0 times

Character 'e' appears 3 times

...

**1.2 Focus**

Coding mechanics

**1.3 Solution**

Anything that iterates through the characters in a string and increments it's frequency.

* Must count uppercase as lowercase
* Can focus on english alphabet, Bonus for handling any Java Character, Character.isLetter
* Result is lexicographically sorted

**1.4 Common Mistakes**

**1.5 Sample solution (Java)**

...  
**public** **static** int[] countOccurrenceOfLetters(String string) {  
 int[] chars = **new** int[26];  
 **for** (int i = 0; i < string.length(); i++) {  
  char c = string.charAt(i);  
  **if** (c >= 'a' && c <= 'z') {  
    chars[c - 'a']++;  
  } **else** **if** (c <= 'A' && c >= 'Z') {  
    chars[c - 'A']++;  
  }  
 }  
 **return** chars;  
}  
  
**public** **static** void main(String[] args) {  
 int[] chars = countOccurrenceOfLetters("Amazon Web Services");  
 **for** (int i = 0; i < chars.length; i++) {  
    System.out.println("Character '" + (char)(i + 'a') + "' appears " + chars[i] + " time" + (chars[i] == 1 ? "" : "s"));  
 }  
}

**Find sum in array**

**Question (Problem Solving):** Given an array of integers and some value X, find the unique pairs of integers that sum to X.

**Solution:** There are a few solutions to this problem. Any solution that runs in O(n lg n) or O(n) time is acceptable. One way to get O(n lg n) is to sort the array using merge sort or something similar. Then use binary search on each element to see if X minus that element exists in the array. A O(n) solution could involve iterating over each element, checking to see if (X – element) is in a set, then adding the element to the set.

**Common Mistakes**:

* The most common mistake is to give the brute force solution and then not be able to make further progress.
* Not knowing specifics about sorting algorithms. Many candidates will say something like, “Sort the array, “and when you ask how/what’s the best/worst case runtime you get silence. It’s important that candidates know the specifics about the algorithms they want to use.
* A corollary to the above is candidates that want to use a HashTable to solve the problem, but don’t know how Hash Tables work.
* When using the hashing solution, almost no one sees the edge case where there’s a single occurrence of X/2 in the array. If implemented incorrectly (putting all of the elements in the hash table to start with) you’ll end up matching elements against themselves to form pairs. I usually just let this go, but sometimes for SDE-II candidates I check to see if they can fix this bug.

**Analysis**: While this question is all over the internet I find that it still has significant diagnostic value for checking problem solving ability. **It’s very important to dig deep on this question as it can give a lot of insight about the strength of a candidate’s CS fundamentals. Please make sure to probe about sorting algorithms and hash tables as part of this question.**If they can clearly describe a solution that’s faster than O(n^2) and they demonstrate that they know how the data structures and sorting algorithms they want to use work, credit should be given. If they show a lack of understanding for CS concepts or can’t clearly describe a working solution that’s faster than brute force, no credit should be given.

**1 Detect whether a binary tree is a binary search tree**

**1.1 Question (Coding)**

Given the root node of a binary tree of integers, determine if it’s a binary search tree.

**1.2 Focus**

Recursion, tree data structures

**1.3 Solution**

Make sure that you talk through the solution with the candidate first. It’s important that you don’t waste time letting the candidate code an incorrect solution. The simplest solution to implement is recursive. In the recursive call we’ll pass a node, a minimum value, and a maximum value. The base case is checking to see if the value of that node is within the bounds (initial bounds should be MAX\_INT and MIN\_INT). Then depending on whether you recursively go left or right you update either the max value or min value.

If they suggest an in-order traversal and confirm the elements are ascending, then push them to see if they can eliminate the need to store the entire array and do the comparisons as the traverse the tree. O(1) space complexity

**aside** in-order traversal will not work if duplicates are allowed.

5

/ \

4 5

\

5

**1.4 Common Mistakes**

* The most common mistake candidates have is forgetting that a BST’s left **subtree** is less than the root node, not just the left node itself. Because of this many people think that it’s sufficient to check just a single node below a given node to see if it’s less than the current one. This is not sufficient to solve the problem correctly. This mistake is common enough that I don’t really care if a candidate tries this as their first solution as long as they can understand and react appropriately to the feedback.
* Most other mistakes generally have to do with implementing a recursive algorithm incorrectly.

**1.5 Analysis**

It’s not expected that a candidate will be able to immediately get to a recursive Min/Max solution, but with a bit of guidance they should understand where they’ve made mistakes and eventually be able to get to a correct solution. Even if you have to find a way to subtly hint that passing around some bounds that the nodes have to fall within, if the candidate can then figure out the solution and code it up cleanly, credit should be given. If they keep getting stuck and can’t make any progress even with some reasonable hints no credit should be given.

**1.6 Sample Solution (Java)**

**1.6.1 recursive using min, max**

**private** boolean isBST(Node node, int min, int max) {  
**if** (node == **null**)  
 **return** **true**;  
**if** (node.value < min || node.value >= max)  
 **return** **false**;  
**return** isBST(node.left, min, node.value)  
  && isBST(node.right, node.value, max);  
}  
  
**public** boolean isBST(Node node) {  
**return** isBST(node, Integer.MIN\_VALUE, Integer.MAX\_VALUE);  
}

**1.6.2 in-order traversal O(1) space**

**private** Node previous = **null**;  
  
**public** boolean isBST(Node node) {  
   **if** (node == **null** )  
    **return** **true**;  
     
   **if** (!isBST(node.left))  
    **return** **false**;  
    
   // Allows only distinct valued nodes   
   **if** (previous != **null** && node.value <= previous.value)  
    **return** **false**;  
     
    previous = node;  
  
   **return** isBST(node.right);  
}

**1.6.3 Examples**

Happy case

8

/ \

4 10

/ \ \

1 5 12

Failure case (simple)

8

/ \

4 10

/ \ \

1 3 12

Failure case (subtree check)

6

/ \

4 10

/ \ \

1 7 12

**Combinations without repetitions**

*Variations of this question:* [StringPermutations](https://w.amazon.com/bin/view/StringPermutations) [StringSubsets](https://w.amazon.com/bin/view/StringSubsets)

**Question**

Given N symbols, find all possible unique combinations of K symbols from N. The order of the symbols in the tuples is irrelevant.

To make this a little more concrete consider the following:

N = A,B,C

If K=1, the results is A,B,C

if K=2, the result is AB, AC, BC

**Focus**

Problem Solving, Algorithm generation, Abstraction, Coding

**Tips for the interviewers**

* Get the candidate to enumerate for N=ABCD, k=2 and k=3. See if they can see the pattern.
* Get the candidate to use the emerging pattern to define an algorithm
* This can be solved recursively or linearly
  + Most candidates will approach the problem recursively
  + The recursive solution requires less up front thought but its often harder to implement
* Candidate often treat the K symbols as a string, this is harder
  + The symbols are just an enumeration and can be represented as as an arbitrary set of integers, I would expect SDE3 or an good SDE2 to pick up on this without prompting
* Measure how much prompting does it take to get a working algorithm out of the candidate?
* Can they convert their algorithm into code?
* Does the code handle edge cases:
  + K=0
  + N=0
  + K > N

**Linear algorithm**

The total number of possible combinations is: n!/(r!(n-r)!) (! is the factorial)

Before we describe the algorithm consider the following examples. This first uses a combination with tuples of size 2 from 5 possible characters (ABCDE) - expanded for clarity:

AB | AB

A | AC

A | AD

A | AE

A | AF

---|----

BC | BC

B | BD

B | BE

B | BF

---|----

CD | CD

C | CE

C | CF

---|----

DE | DE

D | DF

---|----

EF | EF

Similarly for a set of 3:

A B C

A B D

A B E

A C D

A C E

A D E

B C D

B C E

B D E

C D E

Now a description of the algorithm:

Between combinations we "increment" from the right up to a maximum value and overflow to the left (very similar to how we do "long addition"). When we overflow we reset the values to right of the last overflown value. The maximum value depends on the column position.

**Linear Solution**

* SDE1: Bonus, I would expect them to jump to a recursive solution
* SDE2,3: Looking for a reusable implementation
* SDE3: Looking for defensive programming (protecting internal state)

**package** **com.amazon.test**;  
  
**import** **java.util.Arrays**;  
  
/\*\*  
 \* Class to enumerate all combination without repetition of r elements from n elements.  
 \* (The order does not matter).  
 \*  
 \* @author Caleb Lyness (clyness@amazon.com)  
 \*/  
**public** **class** **Combinations**  
{  
  /\*\*  
   \* Construct an object which can be used to enumerate all combinations  
   \* without repetition.  
   \*  
   \* @param n is the number of items from which all combinations are  
   \*          to be considered.  
   \* @param r is the number of items to picked per combination.  
   \*/  
  **public** Combinations(int n, int r) {  
    **if** (r <= 0 || r > n) {  
      **throw** **new** IllegalArgumentException("The number of element in the combination must be less than or equal to the number of elements available (r <= n, where r > 0)");  
    }  
    **this**.n = n;  
    **this**.r = r;  
    combination = **new** int[r];  
    reset();  
  }  
  
  /\*\*  
   \* Set the internal state back to the first combination.  
   \*/  
  **public** void reset() {  
    exhausted = **false**;  
    **for** (int i=0; i < r; i++) {  
      combination[i] = i;  
    }  
  }  
  
  /\*\*  
   \* Calculate the next combination of r elements from n. If there are  
   \* no more valid element null is returned. Otherwise an array of r  
   \* elements is returned. The result is a defensive copy of the internal  
   \* state and may be manipulated and retained.  
   \*  
   \* @return  
   \*/  
  **public** int[] next() {  
    **if** (exhausted) {  
      **return** **null**;  
    }  
    int[] result = Arrays.copyOf(combination,r);  
    nextCombination();  
    **return** result;  
  }  
  
  /\*\*  
   \* Crux of the problem/algorithm, the rest is mainly plumbing:  
   \*  
   \* Determine the next combination assuming the member combination contains  
   \* the last combination calculated. If all combinations have been  
   \* enumerated the exhausted flag is set to true.  
   \*  
   \*/  
  **private** void nextCombination() {  
    int i = r-1;  
  
    // Increment and 'carry' from right to left.  
    **while**(i >= 0 && ++combination[i] > n-(r-i)) i--;  
  
    **if** (i >= 0) {  
        // Rolled over to the current position  
      // Move in the reverse direction (...,x,x+1,x+2,...)  
      **for** (;i < r-1; i++) {  
        combination[i+1] = combination[i]+1;  
      }  
    } **else** {  
        exhausted = **true**;  
    }  
  }  
  
  **private** **final** int r;  
  **private** **final** int n;  
  **private** **final** long combinationsCount;  
  
  **private** boolean exhausted;  
  **private** **final** int[] combination;  
  
  **public** **static** void main(String[] args) {  
    Combinations combination = **new** Combinations(20,3);  
    int[] result;  
    **while** ((result = combination.next()) != **null**) {  
      StringBuffer buf = **new** StringBuffer(result.length);  
      **for** (int i=0; i < result.length; i++) {  
        char c = (char)('A' + result[i]);  
        buf.append(c);  
      }  
      System.out.println(buf.toString());  
    }  
  }  
}

**Recursive Solution (Java)**

There are multiple recursive solutions but the optimal would be to treat the problem space as a tree and you are just doing a depth first traversal (to depth k) of all the remaining options.

A ... B C D E

/ |\ \ /|\ |\ |

/ | \ \ C D E D E E

/ | \ \ /| | |

/ | \ \ D E E E

B C D E

/|\ /|\ |\

C D E D E F E F

*Guts of recursive solution.. no error checking or short branch elimination.*

**public** **static** void makeCombination(String prefix, String remainder, int k) {  
  
  **if** (prefix.length() == k) {  
    System.out.println(prefix);  
    **return**;  
  }  
    
  **for** (int i = 0; i < remainder.length(); i++) {  
    makeCombination(prefix + remainder.charAt(i), remainder.substring(i + 1), k);  
  }  
}  
  
**public** **static** void main(String[] args) {  
  makeCombination("", "ABCDE", 3);  
}

## 1 Find nearest 100 points

### 1.1 Question (Problem Solving)

Find the 100 points closest to the origin in a 2D graph.

### 1.2 Focus

Complexity analysis, heap data structures or binary search

### 1.3 Solution

Read each element. Calculate its distance from the origin. If the set had fewer than 100 elements, add it. If it has more than 100 elements, but is smaller than the largest element, remove the largest element, add the new smaller element. Repeat until all points have been accounted for.

This one is fairly straight forward. You’ll need to maintain an ordered list of some type, access it intelligently and prune it as necessary. Now, the easiest way to do this is to use a max heap and remove an element for every element over the 100th that’s added. However, specific knowledge about max heaps isn’t required. You could do the same thing with an ordered list of times that’s bounded at a size of 100 elements. Any solution that runs in O(n lg m) time is acceptable. I’d even accept a solution that runs in O(nm) time if they can justify that the m is a known constant. Obviously, the candidate will need to also be able to calculate the distance from the origin. If you want a fun twist you can tell the candidate that they don’t have access to any math packages and they can only use addition, multiplication, division, and subtraction. Note: We just want the 100 points closest to the origin; since we don’t ever actually have to calculate the actual distance from the origin (just the relative distance), you don’t need a square root function to solve this problem.

### 1.4 Common Mistakes

There are some candidates that don’t understand the problem and will try to use O(n) space instead of O(100) space. They should be told that optimising for space here also helps to optimise for time. Some candidates get hung up on calculating the actual distance from the origin and use some new invented math. I’m fine with anything that actually works here, but ask candidates to explain their design decisions.

### 1.5 Analysis

Again, we’re simply looking for candidates to get to a solution that runs in a reasonable amount of time. If they can get to the O(n) solutions to this problem with a reasonable amount of guidance they’ve passed the question. If they can get the solution, but not do the runtime analysis, you’ll need to make a call as to whether they’re just have problems with runtimes on this problem or if there’s a serious gap somewhere in their knowledge. We shouldn’t bring any candidate in house that can’t do some level of runtime analysis (they don’t need to be absolutely perfect, but close to perfect is desirable). If they can’t get to a solution that runs in the right time or space they’ve failed the question.

**Find first single char in String**

**Question (Problem Solving)**

Given a string of characters, find the first character that occurs only once.

e.g. supercalifragilisticexpialidocious --> f

**Focus**

Data structures, Complexity

**Solution**

There are many solutions to this problem. Generally, if the size of the language is not dependent on the size of the string (n) a linear solution should be the goal here. People that are really in to Java might answer that they’d use a TreeMap with O(n lg n) time, or they might say they’d use a LinkedHashMap. Both of those answers are acceptable, but the next question should be, “Implement a TreeMap or LinkedHashMap.” However identifying that you need to keep track of order and count is probably about 1/3 of the question, seeing that you need 2 data structures of sorted tree to accomplish this in a reasonable amount of time is another chunk.

My favorite answer is one that uses a queue and a hash table to keep track of things. First check and see if the element exists in the Map, if not, add it to the map with a count of one and add it to the head of the queue. If it does exist, add one to the count. At this point you can potentially remove it from the queue if you like. Whether that’s an actual real-world optimisation depends on the nature of the input data set. When you reach the end of the stream, simple pull elements off the end of the queue until you find something with a count of 1. If you were removing as you went along your answer is the last element in the queue. As some extra credit for this question I like to see if the candidate can give big-O for how long it takes to search through the list they’re maintaining. Since the maximum size of the list is related to the size of the character set, which is constant, it’s a constant time operation, not an O(N) operation like most people are used to. This has the potential to be very diagnostic about someone’s ability to do runtime analysis.

**Common Mistakes**

* Many people will try to store the entire stream and do a multi-pass solution. This isn’t that big of a deal as long as they can make good progress after you point out that there isn’t necessarily enough memory to keep the entire stream in memory (hint hint).
* Lots of candidates completely forget about the “first character” part of the question and simply implement a solution that uses a hash table to keep count. Again, that type of mistake is generally ok if they can recover from it without excessive hints.
* Finally, some people like to prematurely optimize and want to keep a list of characters and simply add and remove characters from the list? Example, look and see if a character is in the list, if it is, remove it because it’s occurred more than once, if not, add it to the list. What you end up with here is a list of characters that occur an odd number of times, not single occurrences. This should be caught by making a candidate work through an example.

**Analysis**

This question can prove to be a little bit difficult for some candidates to understand, try to make sure the candidate really understands what’s being asked before letting them run with it. For this one any working solution should be good enough. There are lots of ways to solve this. If they decide to use a more complex data structure, make sure that they could build one in a pinch and know all the runtime details. If they’re rolling their own solution generally try to get them to a solution that isn’t unnecessarily complex. This question ends up being fairly binary, some candidates will get it and some will be lost from word one.

### Problem

Given a string, return the longest repeated substring.

#### Examples

* "banana" -> "ana"
* "tomato" -> "to"
* "aaaaaa" -> "aaaaa"
* "Ask not what your country can do for you, ask what you can do for your country." -> " can do for you"

#### Clarifications

* Is this word based? **No**, only think in terms of characters
* Does case matter? **Yes**, but what if it didn't?
* Can substrings overlap? **Yes**, refer to banana

#### Solutions

There are a wide range of approaches, commonly ranging from O(N^4) to O(N). The nice thing about this problem is one solution should lead to another. I've written the code in Java, since the approach may seem overly dependent on C style strings. However, some very nice aspects of Java can be utilized.

#### Very Naive

Simply compare all possible substrings. This is O(n^4) because 3 loops with subString and equals cost O(n).

**static** String findVeryNaive(String s) {  
    String maxCommon = "";  
    int max = Integer.MIN\_VALUE;  
  
    **for** (int len = 1; len < s.length(); len++) {  
        **for** (int i = 0; i < s.length() - len; i++) {  
            **for** (int j = i + 1; j < s.length() - len + 1; j++) {  
                **if** (s.substring(i, i + len).equals(s.substring(j, j + len))) {  
                    **if** (len > max) {  
                        maxCommon = s.substring(i, i + len);  
                        max = len;  
                    }  
                }  
            }  
        }  
    }  
    **return** maxCommon;  
}

#### Less Naive

The observation that should lead to this approach is that length needs not be explicit. Given two character indicies in a string, you can find the longest similarity they represent.

**static** int comSubStringLen(String s, int a, int b) {  
    int i = 0;  
      
    **for** (; a + i < s.length() &&   
         b + i < s.length() &&   
         s.charAt(a + i) == s.charAt(b + i); i++);  
      
    **return** i;  
}  
  
**static** String findNaive(String s) {  
    String maxCommon = "";  
    int max = Integer.MIN\_VALUE;  
      
    **for** (int i = 0; i < s.length() - 1; i++) {  
        **for** (int j = i + 1; j < s.length(); j++) {  
            int com = comSubStringLen(s,i,j);  
              
            **if** (com > max) {  
                maxCommon = s.substring(i, i + com);  
                max = com;  
            }  
        }  
    }  
    **return** maxCommon;  
}

#### Sort-based Solution

This solution runs in NLogN time, with a sort followed by a linear time traversal of the sorted list (*looks like N2 log N to me, would someone care to elaborate?*). This solution arrives from the understanding that substrings that have the most in common also appear next to each other in sorted order.

This algorithm is [described in some lecture notes](http://bozeman.mbt.washington.edu/compbio/mbt599/Lecture1/template.html) which claim it is O(n\*log2n) "average case".

nhandao@: this looks more like O(n^2) to me, because the last loop has equals() which is O(n).

@junma: I do not believe this could be NLogN. The sort will be NLogN by itself. The compare function is O(N). The total sort time is already NLogN \* N

**static** int comSubStringLen(String s, int a, int b) {  
    int i = 0;  
      
    **for** (; a + i < s.length() &&   
         b + i < s.length() &&   
         s.charAt(a + i) == s.charAt(b + i); i++);  
      
    **return** i;  
}  
  
**static** String findNLogN(**final** String s) {  
    String maxCommon = "";  
    int max = Integer.MIN\_VALUE;  
      
    List<Integer> stringIndicies = **new** ArrayList<Integer>(s.length());  
      
    **for** (int i = 0; i < s.length(); i++) {  
        stringIndicies.add(i);  
    }  
      
    Collections.sort(stringIndicies, **new** Comparator<Integer> (){  
        @Override  
        **public** int compare(Integer a, Integer b) {  
            **for** (int i = 0; i + a < s.length() && i + b < s.length(); i++) {  
                **if** (s.charAt(i + a) != s.charAt(i + b))  
                    **return** (int)s.charAt(i + a) - (int)s.charAt(i + b);  
            }  
            **return** 0;  
        }  
    });  
      
    **for** (int i = 1; i < s.length(); i++) {  
        int com = comSubStringLen(s, stringIndicies.get(i), stringIndicies.get(i - 1));  
        **if** (com > max) {  
            max = com;  
            maxCommon = s.substring(stringIndicies.get(i), stringIndicies.get(i) + max);  
        }  
    }  
      
    **return** maxCommon;  
}

### String rotation solution

The idea is to rotate the string N-1 times and compare each rotation with the original string to find the longest \*aligned\* substring. The longest of all these substrings is the solution. The reason we rotate N-1 times is because the Nth time will return the original string which will match with itself.

Rotations can be done easily in O(n^2): O(n) strings, each costs O(n) to create.

Longest \*aligned\* substring example: "ABC12345D" and "ABCD12345", the longest aligned substring is "ABC" while the overall longest substring is "12345". This can be done in O(n) as below:

   **private** **static** **class** **Match** {  
        **final** int start, end, length;  
  
        Match(**final** int start, **final** int end) {  
            **this**.start = start;  
            **this**.end = end;  
            **this**.length = end - start + 1;  
        }  
    }  
  
    **public** **static** String getLongestAlignedSubString(**final** String a, **final** String b) {  
        **final** char[] aArr = a.toCharArray();  
        **final** char[] bArr = b.toCharArray();  
  
        int startIndex = 0;  
        int endIndex = 0;  
        boolean ongoingMatch = **false**;  
        **final** List<Match> matches = **new** ArrayList<>();  
  
        **for** (int i = 0; i < aArr.length; i++) {  
            **if** (aArr[i] == bArr[i]) {  
                **if** (ongoingMatch) {  
                    endIndex++;  
                } **else** { //first char matching  
                    startIndex = i;  
                    endIndex = i;  
                    ongoingMatch = **true**;  
                }  
            } **else** { //not match  
                **if** (ongoingMatch) { //add previous match  
                    ongoingMatch = **false**;  
                    matches.add(**new** Match(startIndex, endIndex));  
                }  
            }  
  
            **if** (ongoingMatch) { //add last match  
                matches.add(**new** Match(startIndex, endIndex));  
            }  
        }  
  
        **final** Match longestMatch = matches.stream().max(Comparator.comparingInt(match -> match.length)).get();  
        **return** a.substring(longestMatch.start, longestMatch.end + 1);  
    }

Looping through these substrings to get the longest one is O(n), overall O(n^2).

### Suffix-based Solutions

Suffix trees are an extremely efficient data structure for detecting patterns within a string.

Given a target string S, the general idea is to construct a tree where:

* Edges are labeled with non-empty strings.
* Every path from the root to the leaves is a unique suffix of S
* The tree contains all suffices of S
* Internal nodes have 2 or more children.
* A special character (usually $) indicates the end of string.

For example, here's the suffix tree for [Banana](http://en.wikipedia.org/wiki/File:Suffix_tree_BANANA.svg).

Suffix trees have many useful properties. In particular, the path to any internal nodes represents a prefix common to 2 or more suffices: in other words, a substring that is duplicated within S. To find the longest duplicated substring, we search for the internal node whose path has the largest number of characters. It's relatively easy to convince yourself that this node will be the parent of a leaf node. It turns out that it will be the internal node with maximum depth (this is non-trivial to prove).

Assuming that we already have a suffix tree for our string, the solution is reduced to a tree-traversal. In the worst case, every letter in the string is the same. This will produce a suffix tree of height N + 1, so the search will be O(N).

The trickier part is generating the suffix tree. It's easy to come up with a O(N^2) approach:

SuffixTree tree = **new** SuffixTree();  
  
**for** (int i = 0; i < target.length; i++) {  
    String suffix = word.substring(i, word.length);  
    tree.add(suffix);  // Worst-case: Compare every letter in the tree.  O(N).  
}

It's non-trivial to get faster. [Ukkonen's algorithm](http://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english/9513423#9513423) can produce such a tree in O(N) time; however, this algorithm is very complex and even an expert will not be able to explain nor prove its correctness within an interview time frame.

If it takes O(N) to construct the tree and O(N) to traverse it, the final solution will be O(N).

##### Related solutions

A similar solution is to use a suffix-array data structure. This is basically a suffix tree stored using an array data structure. It solves the problem in a similar way, but has better space requirements and cache access properties when the target string is very long (ex. DNA sequences).

##### Note

Suffix data structures are not widely known, even within Masters level graduates of Computer Science. While its possible a strong candidate might be able to come up with this approach on their own, if they rapidly choose a suffix-based solution they likely have domain expertise. Make a note of this and then ask a different (non-string related) algorithm question.

Draw the following system: A website with multiple webservers connected to a single relational database on a third server. Like this:

+------------+ +------------+

| web server | | web server |

+------------+ +------------+

+----------+

| database |

+----------+

Your manager comes to you one day and says "The site is slow." Your job is to investigate what is going wrong. This is very relevant to what [SDEs](https://w.amazon.com/bin/view/SDEs) at this company frequently do. It is also very open-ended and there are many acceptable responses.

In general, you are looking for the candidate to attack the problem, compartmentalize the subsystems. This also allows you to see how much depth they have in a variety of problem domains, from networking to databases to distributed systems.

Good answers:

* Connect to the site with a web browser and verify that it actually is slow.
* Connect to each web server individually to determine if it is a problem with only one web server.
* Look at the logs. *People who expect logs are people who are used to writing maintainable software.*
* Run a test suite to see where there's trouble. *Similar to looking for logs, this suggests that they have good practices for writing robust, maintainable software.*
* Look at web server logs to validate that each web server is receiving an appropriate share of the traffic. If not, something could be messed up with the upstream router/load-balancer.
* Look at both the web server and database server at a system level. Are the machines constrained on disk I/O, memory, processor? Make sure you get specifics from the candidate as to which tools they would use.
* See if the network pipe between the database server and the web server is flooded, and needed to be expanded.
* Is there some sort of database connection pooling going on? Is it sufficient?
* Take a look at the SQL statements to verify that they weren't doing anything that would thrash the server. If so, tune the statements.
* Look into the [DB](https://w.amazon.com/bin/view/DB) schema, to see if there were proper indices.
* Look at the application that accesses the database: is it locking up a whole table for each insert? Is it hitting the database only when it needs to, or is it hitting the database for every page load?
* Verify if any UI script or [HTML](https://w.amazon.com/bin/view/HTML) that is causing a long render time.

Terrible answers (these are real):

* We should just buy faster hardware. ("what if the hardware is already top of the line?") then it's fast enough.
* I wouldn't trust any business owner to tell me that the system wasn't performing well enough.
* Ask the system administrator to figure it out.

Once you are satisfied with their analysis, here are follow-up questions:

*It turns out that the site got Reddited (old school: dugg; old old school: slashdotted). The traffic is immense and the bottleneck is the database (the web servers are largely idle waiting for the database to finish returning records.) What can be done in software to make the site more speedy?*

If the database is the scarce resource, you are looking for the candidate to propose some sort of caching scheme. In addition to seeing what the candidate proposes to cache (raw database records, "objects", web pages, etc), there are lots of drill-down questions regarding how the cache will work. For example: where will the data be cached? Will it be a write-back or write-through cache? The design of a cache depends to a large extend on the characteristics of the data: an ideal cache for read-only data will look quite different from one for read-write data, and likewise for relatively static data vs. real-time or constantly updated data. Once the candidate has fleshed out a cache design, vary the characteristics of the data and see how he/she modifies the cache design to fit.

*This site sells products (what an idea!) and the database contains product records. Your sales department has landed a contract which will add 5 million products to the database. Without adding hardware, what needs to change in the design?*

Retrieving individual product information should not be any different if there are 10 products or 10 million. However, the "search" feature of the website will be affected. See how the candidate makes that conclusion and what ideas they have for, possibly, separating the implementation of searching for records from retrieving them (like what we do.) Another idea is to change the web interface to require certain constraints to searches (such as by price) to avoid a full table scan. Another idea is to generate database views based on product type (shoes, books, etc) again with the express purpose of limiting full table scans of the enormous product table.

See [InterviewWordCountAnswer](https://w.amazon.com/bin/view/InterviewWordCountAnswer) for sample answers.

*On the whiteboard, please write a version of the unix utility "wc" in the language of your choice.*

This is a rudimentary programming question, but it tests many things:

* **Working under stress.** It requires the candidate to actually get up to the whiteboard and write some code under observation. This tests how well they work under pressure, and how well they can "think on their feet", literally. :-) I make sure to not speak at all while they're writing code, unless they ask me a question. It's best to let them sweat it out in total silence.
* **Basic programming abilities.** Any competent programmer should be able to bang this out with very little difficulty. if they can't do this, then you have to wonder how well they know these languages that they say they know. If they try to write pseudo-code, you can press them to actually write real syntax. However, if they forget the exact name of a specific java method call in a specific class, that's no big deal.
* **Understanding requirements and following instructions.** See how well they can reverse-engineer the feature set of wc. If they aren't familiar with wc, then you can describe it to them. See how easily they pick up on your instructions.

Once they've written it one way, I usually ask them how they can implement it a different way. This forces them to "think outside the box" and be creative.

Another variation: once they've solved it, you can add a requirement that the program should report the number of *distinct* words in the file. In addition to introducing a different data structure, this will also test their ability to refactor their code, even if it is just a little.

Added by knox: I also am watching to see them use a Finite State Machine. If they don't (which you will be able to recognize by the complete rat's nest of if statements), it is a very hard problem to code and get right. If they do, it falls out fairly quickly. I also look at how well they handle edge cases: For example, leading or trailing spaces, or lines that consist entirely of spaces, affecting their word count. Bonus points if the first thing they do is to write a simple test case, that they continue to use to evaluate their code.

Added by olivwong@: It turns out I was asked this question when I interviewed at Amazon, and my solution didn't involve a state machine and did not contain a rat's nest of if statement. It was "pseudo-Java", but the basic idea was to use [the built-in split() method](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#split(java.lang.String)) to break the string into words, and then count the words: (This solution does not work for non ascii or other chars like ')

public int wc(String input) {

String[] words = input.split("[^a-zA-Z0-9]+")

return words.length;

}

*How do you think the "People who have shopped for this item also shopped for" feature works on our site?*

Another open-ended question with a lot of possibilities for follow-ups. Some aspects include:

* the scale at which this data operates. millions of customers and millions of orders.
* due to the scale, does the candidate really believe that this data is being queries out of a relational database in real-time? so how is the data stored? how is it generated? how frequently does it need to change?
* the source of the data. is it sessions, orders, both, neither?
* how come everything doesn't recommend Harry Potter?
* How will your system survive during a system crash? (Say perhaps the AZ serving your relational database is suffering from an power outage)

Perhaps some of the Personalization gurus can chime in with other interesting aspects of this question.

Given an array of integers of length n, containing values between 1 and n-1, find the duplicate entry.

Example Array:

[2 1 2]

First let them find a generic solution, then ask them for the fastest solution, and the solution which takes up no extra space but lets you destroy the array. (use it as storage basically).

Note that this problem can be provably reduced to sorting. This means that if a linear time comparison based algorithm is found then we would have one for sorting as well (this is provably impossible). Ergo any linear time solution must make use of some space. Conversely the best algorithm using only comparisons will be to sort the entire list and look for the pair of numbers that are the same.

[**NB**](https://w.amazon.com/bin/view/NB)**:** A solution exists that runs in linear time with constant space. If you view the array values as pointers, the problem becomes equivalent to finding a cycle in a linked list. Here is an example implementation: [DuplicateEntries/Example](https://w.amazon.com/bin/view/DuplicateEntries/Example).

One solution (I have seen a few) for the no extra storage problem (but destroys the original) is to travel to the first element in the array, check its value, then go to that index, and set it negative, now use the value at that index, and skip to that index, again setting it negative. Once you reach an index which is already negative, then you've found your duplicate.

In the example above, you'd go to the first element, find the value to be 2, so skip to the 3rd element in the array (index of 2) which you'd set negative. Now you'd look at that value (-2), and skip to that index (negating it of course), only to find yourself back where you were, and negative, so the repeated value is 2.

Another possibility is, if you take the "the" in "the duplicate" literally, you can sum the numbers in the array and subtract n \* (n - 1) / 2. The difference will be the duplicate entry. This solution uses no extra space, runs in O(n) time (linear), and does not destroy the array. (The sum of the first n - 1 integers is n \* (n - 1) / 2.) This solution assume that there is one and only one dup.

[**NB**](https://w.amazon.com/bin/view/NB)**:** There are some subtleties to the first solution that wait to trap the unwary. The key point is to remember that the arrays are **zero-based** while the number range is from **[1..n-1]**. If you treat the arrays as **one-based** you will find that cases can be constructed where it does not work.

It should also be noted that you can sort the array in place using xors to swap the elements (so you wouldn't be using any addition space). Then, you just return the first collision as the duplicate element.

Java Implementation: <https://w.amazon.com/index.php/User:Arman/Interview#12._ASCII>

This question is either a warm up question (for good candidates) or a weed out question (for bad candidates). It has the advantage of being easy to ask over the phone as well.

Just ask the candidate to write code that emulates the standard atoi function. If they are unfamiliar with the function, a red flag should go up (for C/C++ programmers at least), and then you should explain to them the semantics.

When they are coding the function, you should pay attention to at least the following:

* Do they correctly handle termination conditions? A good candidate will ask for specifics. All candidates should be on the look out for the end of string.
* Do they correctly handle edge cases: empty strings, strings with no digits, [NULL](https://w.amazon.com/bin/view/NULL) strings, etc?
* Do they look for negative numbers and handle them correctly?
* Do they do any tokenization (the standard atoi skips leading white space, for instance)?
* Did they make the input argument const (as it should be)?
* Do they correctly handle input with non-numerical characters? (for example atoi returns 12 if the input is "12a34") If they don't do it at first, can they handle it when you remind them that this is how atoi should behave?
* Do they correctly convert characters to numbers? (e.g. '9' - '0' = 9)

Good candidates should be able to write this out in a matter of minutes. Its a bad sign if a candidate gets stuck on this question. If they get it done quickly, it might be worth spending some time asking harder questions about the problem:

* How would you rewrite the function to handle errors (the standard function doesn't do any error checking)?
* What sort of errors would you look for? Good answers include: strings with no digits and integer overflow (this one's fun to pursue, to see what they can do with it).
* Can you make your implementation more efficient?
* How can you add support for bases other than 10?
* Re-implement the function as a small state machine (this is also a fun one).

Here is some code for a very barebones implementation of atoi:

(Q: How would this code sample handle the string: '123-43-6-7'? Is this the same as atoi?)

int atoi(const char\* s)

{

int val = 0;

int is\_neg = 0;

if(s == NULL) return val;

while(isspace(\*s)) s++;

if(\*s == '-') {

is\_neg = 1;

s++;

}

while(isdigit(\*s)) {

val \*= 10;

val += \*s - '0';

s++;

}

return is\_neg ? -val : val;

}

(Q: Where's the defect in the above sample? A: it doesn't handle a leading '+' sign. --It also doesn't handle MININT, -2147483648.)

A lot of folks will write code that depends on strlen(). They'll try to walk from the right to the left, etc. If so, what's the runtime of the code? It's O(n) if they check the strlen before the loop. It's O(n^2) if they check it in the loop condition. Ask how many times they walk through the string, even if it's O(n). Do they realize strlen() is walking through the string? If they are walking through the string more than once, can they do anything to reduce the number of times they walk through it? The above code only does it once.

Note: this question is discussed in Programming Interviews Exposed: Secrets to Landing Your Next Job ([www.amazon.com/exec/obidos/ASIN/04713...](http://www.amazon.com/exec/obidos/ASIN/0471383562/)), so there's a chance candidates will have seen it before.

Another personal Note: I would ask this question, but then follow up with the question: write code to evaluate a polynomial for any given x (e.g., 4x^7 + 2x^2 - x + 3), and maybe to make it harder and guide the question towards the expected answer, they cannot use the exponent function, or if they write their own brute-force exponent function, ask them to come up with a more efficient algorithm. The reason for this followup question is, if they truly understood and solved the ascii to int problem with the answer they provided on their own rather than via memory or having seen the problem before (and even if they have seen the problem, they should at least have asked, "why does multiplying by 10 work?" because I know I did), then they should be able to do this problem. Horner's algorithm is something that websites that provide the ascii to int solution do not discuss in the slightest (I couldn't find any obvious reference after a quick search) and Programming Interviews Exposed most likely does not go into that either I am willing to bet.

So anyway, a true understanding of ascii to int would be to know that this is simply Horner's algorithm for a simple polynomial of the form x^n-1 + x^n-2 + .... + x^1 + x^0. Where x=10, n=strlen() and the coefficients are the individual digits of the given number. For example 312= 3 \* 10^2 + 1 \*10^1 + 2\*10^0, or 3 hundreds, 1 tens, and 2 ones. But then again, this is more mathematical and I don't expect many people to just come up with this during an in-person interview. But if I were to ever ask this question, I would give extra, extra bonus points to those who point out Horner scheme as that shows that they take ownership to take the time to **actually** understand a problem, rather than just memorize it or 'get it done quickly' or hastily looking up the answer in wikipedia or stackoverflow.

<http://en.wikipedia.org/wiki/Horner_scheme>

This is a warm up question (for good candidates) and a weed out question (for bad candidates). A good candidate should be able to answer this question in a matter of minutes. See [ShuffleArraySolutionsFromInterviewees](https://w.amazon.com/bin/view/ShuffleArraySolutionsFromInterviewees) for some "interesting" solutions.

The question is: given an array of N elements, write a function to permute the elements in the array so that any permutation is equally likely. That means a given element should have a 1/N chance of being in any specific slot of the array. There are a lot of bad approaches to this algorithm, including:

* Randomly picking two numbers and permuting them (for some number of iterations).
* Generating a random index for each element of the array (i.e. for(i=1..N) { r = rand(1...N); a[r] = a[i]; }).
* Create another array of size N then, for each element in the original array, generate a random index and put it in the second array at that slot. If there is already an element in that slot in the second array, keep generating random indices until you find a slot that isn't taken.

The right algorithm to solve this problem is to start with the first slot in the array and choose one of the N elements for that slot. So, for that slot, every element has a 1/N chance of being selected. Then, for the second slot, choose an element from the remaining N-1 elements. In that case, every element has a 1/N-1 chance of being selected, but they also had a N-1/N chance of *not* being selected as the first element and N-1/N \* 1/N-1 = 1/N. The same logic applies for the rest of the elements in the array. This can be done in place by just swapping the current index with the randomly selected index. A good discussion of this, which is also known as the Fisher-Yates shuffle (1938), can be found at <https://en.wikipedia.org/wiki/Fisher-Yates_shuffle>

A cheeky (but great!) answer would be the following:

void shuffleArray(std::vector<int>& v)

{

std::random\_shuffle(v.begin(), v.end());

}

Similarly, in C, there's strfry() function (It's a GNU extension to libc). It works on strings but could be used to permutate indicies in many cases.

A more typical answer (in C) might look something like this:

void shuffleArray(int\* a, int len)

{

int i, r, temp;

// seed the random number generator

srand(time(NULL));

// iterate over each slot in the array and choose an element to put in that slot

for(i=0; i<len; ++i)

{

// pick a random number from i to len-1

r = rand() % (len-i) + i;

// swap a[r] and a[i]

temp = a[i];

a[i] = a[r];

a[r] = temp;

}

}

An idiomatic version, using [STL](https://w.amazon.com/bin/view/STL)'s swap:

#include <algorithm>

void shuffle(int \*a, int len)

{

srand(times(NULL));

while(len)

swap(\*a++, \*(a+(rand() % len--)));

}

[NOTE](https://w.amazon.com/bin/view/NOTE): this solution does not work because the order of evaluation of function parameters is undefined

Bjarne Stroustrup's C++ Style and Technique [FAQ](https://w.amazon.com/bin/view/FAQ) says so [www.research.att.com/~bs/bs\_faq2.html...](http://www.research.att.com/~bs/bs_faq2.html#evaluation-order)

**Limitations**

Note that the Fisher-Yates shuffle is only valid if the random number generator uses a large enough seed value to represent all the possible permutations of the array. For example, a minimum seed size (and random number generator space) of 230 bits is required for a deck of 52 playing cards, since there are 52! = 2^225 possible permutations of the array, and we need at least that amount of different permutations of the prng.

See <http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle#Potential_sources_of_bias> for more details about this and related limitations.

[revised 03/19/2004 - cgordon]

Java Implementation: [User:Arman/Interview#8.\_String\_Permutations](https://w.amazon.com/bin/view/User:Arman/Interview#8._String_Permutations)

This is a good problem to use for warming up or weeding out a candidate. Most good candidates have seen it and done it before and should be able to describe the algorithm fairly quickly. Candidates who haven't seen it before, but have a passing understanding of recursion should be able to get through it relatively quickly. It's a red flag if a candidate gets stuck on this question.

[ScottRu] If they are successful, getting candidates to remove duplicates is a good followup and probably changes their design. If they thought of this along the way - well, they're the first successful candidate I've ever seen do so.

The problem is to generate all possible permutations of a given input string. An efficient solution (from vkostadi) follows. It would be rare for a candidate to come up with this good an answer in an interview (I've never seen it happen in two years of asking this question).

#include <stdio.h>  
   
 void swap(char\* **const** a, char\* **const** b) {  
    char c;  
    c = \*a;  
    \*a = \*b;  
    \*b = c;  
 }  
   
 void permutations(char\* **const** s, int a, int b) {  
    **for** (int i = a; i <= b; i++)  
    {  
       swap(s+a, s+i);  
       **if** (a == b) printf("%s**\n**", s);  
       permutations(s, a+1, b);  
       swap(s+a, s+i);  
    }  
 }  
   
 void main() {  
    char s[256];  
    scanf("%s", s);  
    permutations(s, 0, strlen(s)-1);  
 }

An equivalent functional solution (in Clojure):

(**defn**string-perms-aux [string d len]  
  (**if**(= d len)  
    (println string)  
    (reduce (**fn**[lastswap i]  
              (**if**(= lastswap (get string i))  
                lastswap  
                (**let**[sarr (.toCharArray string)  
                      dchar (aget sarr d)  
                      ichar (aget sarr i)]  
                  (aset sarr d ichar)  
                  (aset sarr i dchar)  
                  (string-perms-aux (String. sarr) (inc d) len)  
                  (get string i))))  
            nil (range d len))))  
  
(**defn**string-perms [string]  
  (**let**[len (.length string)]  
    (string-perms-aux string 0 len)))

If you have a really good programmer as a candidate, you might try asking them to recode the above algorithm iteratively (using a stack). Someone who deeply understands recursion should be able to come up with a solution like the following with a little thought:

#include <string>  
 #include <stack>  
 #include <utility>  
 #include <iostream>  
   
 void permutations(string s)  
 {  
     std::stack<pair<int,int> > state;  
   
     int len = s.size();  
   
     int i = 0;  
     int a = 0;  
   
     **while**(1)  
     {  
         **while**(a < len)  
         {  
             swap(s[i], s[a]);  
             state.push(pair<int,int>(i,a));  
             a++;  
             i = a;  
         }  
   
         cout << s << endl;  
   
         **while**(1)  
         {  
             **if**(state.empty()) **return**;  
   
             pair<int,int> p = state.top();  
             state.pop();  
             i = p.first;  
             a = p.second;  
             swap(s[i], s[a]);  
   
             i++;  
             **if**(i < len) **break**;  
         }  
     }  
 }  
   
 int main(int argc, char\*\* argv)  
 {  
     permutations(string(argv[1]);  
 }

[kseptor] I think a C++ ninja would come up with something like this:

#include <algorithm>  
 #include <iostream>  
 #include <string>  
   
 int main(int argc, char\*\* argv)  
 {  
     **using** **namespace** std;  
   
     string input;  
     getline(cin, input);  
     sort(input.begin(), input.end());  
     **do** {  
         cout << input << endl;  
     } **while** (next\_permutation(input.begin(), input.end()));  
 }

Here's my solution. Unlike the ones above, it doesn't embed the uniqueness logic in the permutation generation -- rather, it's handled by std::set. Less efficient, but arguably more comprehensible (without being trivial like the next\_permutation() solution, though that's my favorite). -- [lukeme](https://w.amazon.com/bin/view/User:Lukeme)

#include <string>  
#include <iostream>  
#include <cstddef>  
#include <set>  
#include <algorithm>  
  
**typedef** std::set<std::string> Permutations;  
  
void getPermutations(Permutations     & permutations,  
                     std::string **const**& prefix,  
                     std::string **const**& letters)  
    {  
        **if** (letters.empty())   
            { permutations.insert(prefix); }   
        **else**   
            {  
                **for** (size\_t i = 0; i < letters.size(); ++i)  
                    {  
                        char        **const** letter   (letters.at(i));  
                        std::string       whatsLeft(letters);  
                        whatsLeft.erase(i,1);  
                        getPermutations(permutations, prefix + letter, whatsLeft);  
                    }  
            }  
    }  
  
void printOne(std::string **const**& s)  
    { std::cout << s << std::endl; }  
  
void printPermutations(std::string **const**& s)  
    {  
        Permutations permutations;  
        getPermutations(permutations, "", s);  
        std::for\_each(permutations.begin(), permutations.end(), printOne);  
    }  
  
int main(int argc, char\*\* argv)  
    {  
        **if** (argc < 2)   
            { printPermutations("foobar"); }  
        **else**  
            {      
                **for** (int i = 1; i < argc; ++i)   
                    { printPermutations(argv[i]); }  
            }  
        **return** EXIT\_SUCCESS;  
    }

Following is a solution based on the very interesting method called Knuth's Permutations. This has the duplicate permutation avoidance logic built-in (Please ping me at sunnya@ in case of any errors/queries). The first call to the method is getPermutations(str,0).

# include <algorithm>  
  
**using** **namespace** std;  
  
void getPermutations(char \* s,unsigned int nxtWorkChar) {  
  
    **if**(s[nxtWorkChar] == '\0') {  
        printf("%s**\n**",s);  
        **return**;  
    }  
  
    getPermutations(s,nxtWorkChar + 1);  
    **for**(int i = nxtWorkChar ; i > 0 && s[i] != s[i - 1]; i--) {  
        swap(s[i],s[i-1]);  
        getPermutations(s,nxtWorkChar + 1);  
    }//for  
      
    **for**(int j = i ; j < nxtWorkChar ; j++)    swap(s[j],s[j + 1]);  
  
}

Not-so-efficient method using Python generators. -- [motalleB](https://w.amazon.com/bin/view/User:motalleb)

**def** perm(l):  
    **if** **not** l:  
        **yield** ''  
    **for** c **in** set(l):  
        l2 = list(l)  
        l2.remove(c)  
        **for** st **in** perm(l2):  
            **yield** c + st  
  
l = list(input\_string)  
**for** p **in** perm(l):  
    **print** p

This is a good test of a candidates [OO](https://w.amazon.com/bin/view/OO) design skills, as well as their ability to communicate their ideas. Its become one of my favorite interview questions, as it has just enough pieces to be interesting, but still small enough to allow good headway in half hour to an hour. This probably isn't a great question to ask over the phone.

The problem is to design a black jack library out of which three different types of programs can be built. The first is a simulator that can play millions of hands and dump statistics based on the games. The second is an interactive game that allows a single player to play against a dealer and (possibly) a couple other computer controlled players. The third is a multiplayer game (over the internet, or locally) that allows humans to play against each other.

I usually start out asking the candidate to design the simulator, as this is the most straightforward (and keeps them from cluttering the issue with [UI](https://w.amazon.com/bin/view/UI)'s and networks). I also like to keep the game somewhat simple to start with. Computer players are allowed to stand or hit on a hand, there is no split, double or insurance. To make things more interesting I usually require that computer opponents be able to choose from a couple of different betting/playing strategies.

Almost all candidates go straight to modelling the classes they want for the library. They usually come up with some or all of the following: Shoe, Deck, Card, Player, Opponent, Dealer, Table, Pot, GameConfiguration, GameHistory, Game, PlayerStrategy, etc. This approach has rarely yielded as good results as the candidates who first write out the logic of a hand of blackjack and work from there:

1. Players make bets.
2. Dealer deals one card to each player face up and one card to themselves face down.
3. Dealer deals one card to each player and themselves face up.
4. Each player is allowed, in turn, to hit until they decide to stand or until they go bust.
5. The dealer plays out their hand according to a fixed strategy.
6. Each player that did not bust has their hand compared to the dealers and gets paid if they win.

Its a design question, so you will get all sorts of different answers from there. My personal favorite is to use an event based approach with a high level "**BlackJackGame**" class that can act as a model in an [MVC](https://w.amazon.com/bin/view/MVC) pattern, or as a simple high level interface for a simulator. Clients can query the model class for statistics and state, or they can request that a hand be started, at which point the model communicates with Player objects (which might in turn communicate with a remote client) to handle their actions.

You can make the game a little more complicated by introducing splits (which require that you keep track of an array of hands for each player). I haven't found that very fruitful, as it only adds bookkeeping to the problem.

*Please walk me through what your object model would be if you had to design a text editor.*

GusLopez asks this question all the time.

A good candidate could go on for the rest of the interview time with just this question. This question can be used to test many things:

* their sense of class design,
* how you separate function from display,
* what primitive data structures you might use to represent textual data. (Flyweight Pattern)
* Explore feature set -- for example, undo -- could be implemented with Command pattern.

This needs more explanation. Somebody document me!

Potentially useful external references:

* <http://www.oodesign.com/flyweight-pattern.html>
* [Flyweight\_pattern](http://en.wikipedia.org/wiki/Flyweight_pattern)
* [Hash\_consing](http://en.wikipedia.org/wiki/Hash_consing)
* [Command\_pattern](http://en.wikipedia.org/wiki/Command_pattern)

*Please design an automated Air Traffic Control system where all the planes talk to a master ATC server which gives them commands that tell them where to fly and when to take off or land.*

This question is good because you can either expand it or constrict it based on the skill of the candidate. It has no wrong or right answer, its purpose is to see how creative the candidate is and how they approach a problem and determine the requirements for a solution.

I often start out by asking them how the planes talk to the server on the ground and what kind of info would they be sending to the server so that it can make the best decisions.

Useful info that the plane is going to want to send to the server:

* plane id (flight # / call sign)
* lat/long coordinates
* altitude
* heading (what direction the plane is flying in)
* speed
* fuel consumption per (second|minute|hour)
* amount of fuel left
* point of origin for the flight
* destination of the flight
* if an emergency is taking place, what kind of emergency is it
* if waiting to land, amount of time spent in holding pattern around airfield
* if waiting to take off, amount of time spent in line waiting to take off

It's interesting to see what kinds of information that they think the planes should pass to the ground based computer. This part of the question is good for prodding them towards pieces of info that they didn't think about and to see how quickly they catch on to your hints.

I let them spend 5-10 minutes designing the client and server classes and what kind of general data members they have. I don't really care about the data types, I just care about what pieces of information they want to track.

Once they've come up with the design for the classes that are used by the plane and the master ATC computer, I ask them about implementation. How do you model the three dimensional position of a plane inside a data structure? What kind of data structure would make position comparisons between different planes efficient? What if you had to track hundreds or thousands of planes at once?

How do you make sure that two planes won't collide? How do you figure out priority when multiple planes want to land? How do you make sure that a plane doesn't run out of fuel while waiting to land? What if one plane wants to take off and another wants to land at the same time?

What kind of model do you use for the communication between the master server and the different planes? Do the planes subscribe to the server when they enter it's zone and then receive broadcasts that direct them (and all the other planes) what to do? Does the server talk to planes individually instead of over a broadcast? Is there any way to make the system be peer->peer only (i.e. no control computer on the ground)?

This question is really open ended, but if you pay attention to their answers, you can drive them towards any part of the problem and see how they approach the design or implementation. It's a great way to check a candidate's skills in certain areas if you're not sure of their skills in that area. Sometimes I spend the whole time discussing the design of the system with the candidate and we don't even get into implementation, and sometimes the candidate ends up spending all their time explaining how they would represent the position of a plane in a data structure and how they would store and log all the data.

Other things you could ask them about:

* message latency between planes and server, is it a problem?
* for certain mission-critical calculations, the space shuttle has multiple computers on board who all attack the same problem. when they find a solution, they compare answers. how might you expand the system so that there are multiple ATC computers in a particular zone? how do the different master computers pass information back and forth? how would they check to make sure that one of their peers wasn't giving out bogus information? what if they figure out that one of their peers is giving out bogus information?

Finally, if they have a [CS](https://w.amazon.com/bin/view/CS) background, I ask them if they see any similarities between the master ATC computer's logic and other processes/methods/systems that are in the operating systems of most computers. Most smart candidates realize that it's a lot like a process scheduler. This problem makes a nice segue into questions about process priority, what queues are, what is a context switch, stuff like that. I only move on to those questions if they have a [CS](https://w.amazon.com/bin/view/CS) background and their resume lists that they did some [OS](https://w.amazon.com/bin/view/OS) work either in school or as part of their job.

*What's the difference between hashing a string and encrypting a string? Give me some real-world examples of using hashing and encryption.*

When you hash a string, you get:

* fixed length output no matter what the length of the original string
* guaranteed one way operation, there is no way you can look at the checksum provided by a hash function and figure out what the original string was because the original string's length doesn't correspond to the hashed value
* no key is used, the only thing applied to the string is a mathematical transformation, so there's no way to get the original text back from the hash string
* collisions

When you encrypt a string, you get:

* variable length output
* mostly guaranteed one-way operation, depends on algorithm and size of keyspace. encryption can be broken with enough cpu time or exploitation of weaknesses in the algorithm or poor key choices
* a way to recreate the original text if you have the proper key

Real world examples of hashing:

* proxy servers hash the url and store the cached content in a directory structure based on the hashed url value:

hash of '[www.amazon.com/exec/obidos/ASIN/15659...](http://www.amazon.com/exec/obidos/ASIN/1565921496)' => f092f0becc648514

store fetched content to /cache/f092/f0be/cc64/f092f0becc648514

When it gets another request for the same [ASIN](https://w.amazon.com/bin/view/ASIN), it hashes the [URL](https://w.amazon.com/bin/view/URL) and sees if it already has content stored for that [ASIN](https://w.amazon.com/bin/view/ASIN) and returns the cached content if it exists.

* unix password files on some flavors of unix

Some unices store the hashed password value in the password file. This is more secure than just using encryption since you can have a 40 character password if you want and it gets hashed to the same size string as a 4 character password. You don't have to hack in support for long passwords since when you try to login, the system just takes the hash of your supplied password and compares it with the hash stored in the password file for your username.

*(Why is this more secure than encryption? The consequence is that someone can break into the system using the 4 character password even if you prefer to use the 40 character version.)*

It is more secure than encryption because the 4 letter password and 40 letter password hash to \*different\* values. The fundamental measure of a hashing algorithm is the chance of collisions. The hash info above said that the passwords hash to the same \*length\* string, not the same string. This means that if you could look at the hash, there's no way to tell the input length from the output length. If someone wanted to type the first paragraph from their favorite book as a password, you wouldn't be able to tell. The encrypted passwords on unices typically are 8 characters or less, which make brute force attacks feasible. you only need to check 93^8 (93 printable characters between ! and ~ in the ascii table) combinations, versus 2^128 or 2^64 for [MD5](https://w.amazon.com/bin/view/MD5) (because you don't know how long the input string is from the hash output), depending on whether you use a brute force attack or a birthday attack (see google).

* checksums on downloads

If you download software that has a signed checksum, you can make sure that the checksum was signed by the software's creator and then compute a checksum based on the file you download. Compare it to the checksum you get from the creator and if they match, you have the exact distribution. If they don't match, you might have a trojan or the wrong version of the distribution.

Real world examples of encryption:

* using [PGP](https://w.amazon.com/bin/view/PGP) to encrypt email
* unix password files (some unices still encrypt passwords instead of hashing them)
* encrypt sensitive business data on your machine
* [ATM](https://w.amazon.com/bin/view/ATM) bank cards
* [SSL](https://w.amazon.com/bin/view/SSL)

I like asking this question because it gives the candidate a chance to fib and guess at an answer, and the creativity of the real world examples they give can show you how they think and how original they are. If they can't think of any real-world examples of encryption or hashing, then that might be a red flag.

Testing knowledge of algorithms

I think there are two camps here A) Give a random question and look at the candidates approaches and how efficient they are. B) Give a specific problem and see if they understand and evaluate

The first one seems to be the most popular here. "Here is a big list of numbers from 1 to 1000, but one is missing -- how will you efficiently find the one that is missing".

The cool thing about problems like this is that there are a variety of correct approaches and the candidate will probably come up with a few of them. Then you can ask the candidate to evaluate the trade-offs of their own approaches. The one thing I don't like about these problems is that the best answer can sometimes be a trick or very specialized knowledge. In these kinds of cases I like to see their thought process along the way. If they don't get the trick, then I explain the trick and see if they can compare the trick with their approaches.

The second one fits into the category of explaining the issues behind a specific method or data structure. The various differences between Hashes and Arrays -- when would you use one over the other. If someone has a [CS](https://w.amazon.com/bin/view/CS) background, I would expect them to know these answers. Also, the "write quicksort" and give the order of it fits in this category. If they don't do well I suggest that they might like to read Algorithms by sedgewick. Please note that not everyone with a [CS](https://w.amazon.com/bin/view/CS) degree will have studied computational complexity of algorithms. If someone can't answer what "order" an algorithm is, I don't hold it against them as long as they can generally describe why one algorithm is better than another.

If the candidate has algorithm background I also like to talk about scaling issues -- what happens with all the data can't fit into memory? What happens when the amount of data you are handling becomes large? The candidate does not actually have to have experience with large datasets, but this does give an indication about their depth of knowledge in the area.

If the person has a numerical analysis background, then I also like to ask questions about sources of error and how to reduce them.

Another good question is, given an array of ints and an int N. Find the location of two elements that add up to N. Now find three elements... There are solutions ranging from N to N2.

Roman Numeral Coding Question

Description: Ask someone to write a program that converts a decimal number to a roman numeral.

Focal categories: Coding Skills

Phrasing: Everyone is familiar with Roman numerals[[citation needed](https://xkcd.com/285/)], but it seems like a lot of people seem to have little differences in the rules. So it's best to establish a baseline:

"M" = 1000

"D" = 500

"C" = 100

"L" = 50

"X" = 10

"V" = 5

"I" = 1

* To find the value of a set of roman numerals you add up the value of the characters.
* A power of ten can only be repeated three times i.e., [XXX](https://w.amazon.com/bin/view/XXX) = 30, [XXXX](https://w.amazon.com/bin/view/XXXX) is not valid.
* Those that are not powers of ten can only appear once, i.e. VV is not valid.
* The numbers must read highest-lowest from the left to the right. (with one exception, see the next rule)
* If a letter of a smaller value appears before a number of a higher value, then the smaller number is to be subtracted from the higher value. ex: [IX](https://w.amazon.com/bin/view/IX) = 9.
* You can subtract only powers of ten i.e., I, X, C
* Only one character can be used to subtract from a larger character. eg IIX = 8 is not allowed.
* You can't subtract a number from one that is more than 10 times greater. That is, you can only subtract I from V or X, X from L or C, etc. For e.g., [IC](https://w.amazon.com/bin/view/IC) can not be used for 99. It must be [XCIX](https://w.amazon.com/bin/view/XCIX).
* I have no idea why they sometimes put IIII on watches. : )
  + *It's (traditionally) to avoid blasphemy -- IV used to stand for "Jove," and later "YHWH." Cite:* [*http://en.wikipedia.org/wiki/Roman\_numerals*](http://en.wikipedia.org/wiki/Roman_numerals)
  + Another explanation that has made the rounds is that IIII provides better symmetry to the watch face than using IV. Though [[[1]](http://www.web40571.clarahost.co.uk/roman/clockface.htm)] provides evidence that in medieval times, IIII was often used instead of IV.

Ranked Solutions:

I think this is a place where the easy solution becomes really the best solution. There is a median to the amount of code you must write to implement the logic versus how much space hard coding values takes and keeping readability. A good coder should be able to discuss these issues.

**Easy:** Can someone figure out what's the max number that can be written with the rules above? It's 3999. If asked, they should be able to figure it out.

Since this is harder than you think, the answer is: MMMCMXCIX = 3999

MMM + CM + XC + IX

MMM = 3000

CM = 1000-100 = 900

XC = 100 - 10 = 90

IX = 10-9 = 9

SUM = 3000+900+90+9 = 3999

MMMDCCCLXXXVIII = 3888 is the naive approach of only adding roman numerals.

MMM + D + CCC + L + XXX + V + III

MMM = 3000

D = 500

CCC = 300

L = 50

XXX= 30

V = 5

III = 3

SUM= 3000+500+300+50+30+5+3 = 3888

Wayne Chiang says:

Another way to think about it: since there's no symbol to represent 5,000, we also cannot represent 4,000. Therefore, the maximum possible value must be 3,999.

**Medium:** They could hard code every value from 1 to 3999. bad. They could hard code every value in the thousands, hundreds, tens and ones. ie:

1 = I

...

6 = VI

7 = VII

...

10 = X

...

60 = LX

70 = LXX

...

100 = C

...

600 = DC

700 = DCC

...

1000 = M

2000 = MM

3000 = MMM

And just keep mod'ing by 10 until and breaking down the number. I like the approach where you have ones, five, and ten. Essentially the pattern is the at each 10^X level is the same. so at the 10^1 level, the ones = X, five = L, and ten = C. At the 10^2 level, the ones = C, five = D, and ten = M.

**Hard:** If you can get them to implement the logic, I think it's pretty good.

**Hints:**

* You can ask them to start off with small numbers. You can start the scope of the problem with "a number between 1 and 10.
* Some people will think they have to implement all the logic for this and not hard code any values. (ie. no lookup table). This can lead to really weird if cases. See if they can get themselves out of it. Try to simplify it for them.

Indicators:

* This question is more of how well they can organize their thoughts. If they can get the logic down, it's a good sign.
* If their "if" cases seem really outrageous, like "if (i < 17 and i > 15) then and they can't get out of that mode, it's a bad sign.

From Andy Chiu

Eric Docktor says:

There are different strategies for implementing the code. I've ranked them here in the order I prefer for reasons described. I don't think it's important that candidates (or you, dear reader) agree with me on which solution is best, only that folks can talk about the tradeoffs of the solutions.

1. Pattern map:

* Validate that the number is <= 3999
* While loop that pulls off the digits from the input number from greatest (1000s digit) to least (1s digit)
* Feed that digit to a function that returns the roman numeral string for that number given the number, and which digit it is. The function should use a pattern map to convert the digit to a roman numeral string. For example, you can use an array of (undef, A, [AA](https://w.amazon.com/bin/view/AA), [AAA](https://w.amazon.com/bin/view/AAA), [AB](https://w.amazon.com/bin/view/AB), B, [BA](https://w.amazon.com/bin/view/BA), BAA, BAAA, [AC](https://w.amazon.com/bin/view/AC)) and then get the pattern for the value by looking up array[value]. Then substitute the A/B/C with the appropriate roman numeral letters for the given position (1s: I, V, X; 10s: X, L, C; 100s: C, D, M; 1000s: M, undef, undef.)
* Append the returned string to the main string within the while loop.

2. If/thens, with while loop:

* Similar to above, but instead of using a pattern map within the function, you find if the value is a 4, 5, or 9, and print the string appropriate to that number. If the value is less than 4, use a for loop to print the right # of Is, Xs or Cs. Else, it's 6-8, so print a V, L or C and then use a for loop to print the Is, Xs or Cs.

3. Hard-code each value between 1-3999 to its roman numeral string

* Would take 3999 assignments, which might take the candidate a little while to write in an interview. On the plus side, the run-time efficiency is O(1), which is better than the above two.
* [poertel]: Nah, they're all O(1) since the problem has a finite domain. The subtleties of big-O notation obviously aren't that important to an engineer, but that's the answer in case you want to know whether your candidate paid attention in algorithms class!
  + *Is that also the case for all algorithms whose time complexity depends on the size of an array? Arrays can't have more than 2^(8\*sizeof(size\_t)) elements.*
    - Why not longer arrays than that? a) size\_t has been and will be altered in time. b) array don't have to always be indexed by a size\_t. c) arrays don't actually have to be all in memory, they can be streamed in.
    - There is a theoretical upper bound of how many atoms exist in the universe. Said bound is a finite number. This however does not mean we can enumerate all atoms in O(1)!
    - BigO notation is meant to show how performance is impacted as the input increases. The fact that the problem has a finite domain does not impact it. You need to compare romanNumeral(1), romanNumeral(2), etc. and watch the runtime in each. For this case (assuming you use a hash table), you'll notice that the runtime is constant. In the others, you'll notice that the runtime increases as the input increases.

Jim Larson says:

Sometimes IIII is used instead of [IV](https://w.amazon.com/bin/view/IV) for superstition: [IV](https://w.amazon.com/bin/view/IV) is the start of the name of IVPITER (Jupiter, or Zeus in Greek), the king of the gods. For bonus points, have the routine consult a configuration file for names of dieties which must be avoided.

Jacob Cohen says:

There's an approach that doesn't depend on what the value of each digit is. It's basically a modification of the greedy algorithm where you subtract the largest remaining value you can. The modification involves letting it compose a smaller power of ten in front of the current value as a subtraction, and adding the compound value if possible.

bool isPowerOfTen(int x) {

while (x > 1) {

if (x % 10) return false;

x /= 10;

}

return x == 1;

}

std::string integerToRoman(unsigned int n) {

std::string result;

struct RVal {

char roman;

int val;

} rvals[] = {

{ 'M', 1000 },

{ 'D', 500 },

{ 'C', 100 },

{ 'L', 50 },

{ 'X', 10 },

{ 'V', 5 },

{ 'I', 1 } };

int rvalCount = sizeof(rvals) / sizeof(rvals[0]);

for (int i=0; i<rvalCount && n; ++i) {

// Basic greedy algorithm, subtract the biggest remaining

// value we can until the value would go negative.

while (n >= rvals[i].val) {

result += rvals[i].roman;

n -= rvals[i].val;

}

// But here's the trick. See if we can add a smaller power of

// ten and get a result large enough to subtract the current

// value.

for (int j=i+1; j < rvalCount; ++j) {

if (isPowerOfTen(rvals[j].val)) {

if (n + rvals[j].val >= rvals[i].val) {

result += rvals[j].roman;

result += rvals[i].roman;

n -= (rvals[i].val - rvals[j].val);

}

break;

}

}

}

return result;

}

Ben Olmstead says:

I've normally implemented int->roman like:

std::string integerToRoman(unsigned int n) {

std::string result;

struct RVal {

const char \*roman;

int val;

} rvals[] = {

{ "M", 1000 },

{ "CM", 900 },

{ "D", 500 },

{ "CD", 400 },

{ "C", 100 },

{ "XC", 90 },

{ "L", 50 },

{ "XL", 40 },

{ "X", 10 },

{ "IX", 9 },

{ "V", 5 },

{ "IV", 4 },

{ "I", 1 } };

int rvalCount = sizeof(rvals) / sizeof(rvals[0]);

for (int i=0; i<rvalCount && n; ++i) {

// Basic greedy algorithm, subtract the biggest remaining

// value we can until the value would go negative.

while (n >= rvals[i].val) {

result += rvals[i].roman;

n -= rvals[i].val;

}

}

return result;

}

Especially for the rules given here, it's easier just to write out the few IX-style cases.

Andrew Evenson says:

The problem requires some amount of information, probably saved in a structure, and a routine that utilizes the information to build the answer. I look for a good balance about how much information to store against how much code has to be written to use it. I also look for clear thinking on the algorithm that builds the solution. I also prefer if 'i' can be manipulated arithmetically rather than converted to a string first. I also like when a candidate can explain his/her choice to encapsulate (e.g. StringBuilder) over tune (e.g. write chars into a buffer and track the offset).

private static final String[][] simple\_table = new String[][]{

new String[]{"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"},

new String[]{"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"},

new String[]{"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"},

new String[]{"", "M", "MM", "MMM"}

};

public static String toRoman (int i) {

if (i < 1 || i > 3999) throw new RuntimeException("Number " + i + " is not in range [1...3999]");

StringBuilder b = new StringBuilder(16);

for (int offset = 0; i > 0; i /= 10) {

b.insert(0, simple\_table[offset++][i % 10]);

}

return b.toString();

}

If you have a lot of digits to convert and you want the best performance, you can always build an expanded table using the method defined earlier. For this reason, I would not give any points for someone who hand wrote the entire table.

private static final String[] expanded\_table = buildExpandedTable();

private static String[] buildExpandedTable(){

String[] table = new String[4000];

for (int i = 1; i < 4000; i++) {

table[i] = toRoman(i);

}

return table;

}

public static String getRoman (int i) {

return expanded\_table[i]; // returns null for '0' and out of range for all others...

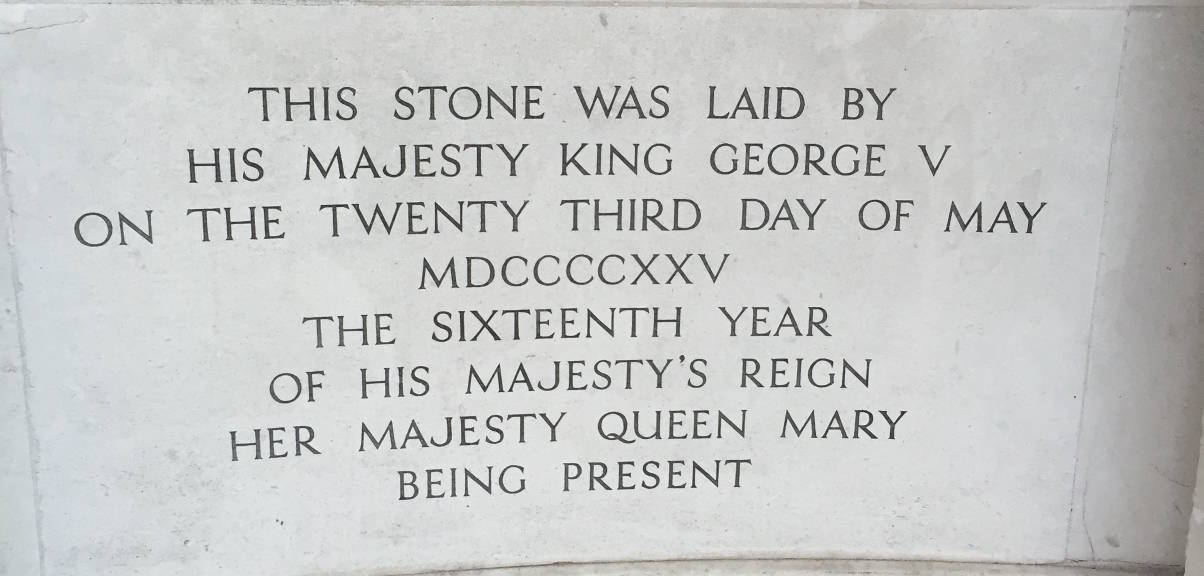
}

Chris Weight says says:

Note that the rules above allow you to create 4,000 also (just add a trailing I, thus MMMCMXCIXI), but since 4,000 is actually represented as IV where V has a bar over it to indicate 5,000 (5 \* 1,000), then one would not use MMMCMXCIXI.

Michael Holst (mschlage@) says:

[Here](http://tiny/q8krjwdg/paste) is a Java implementation I did similar to Andrew's.

[](https://improvement-ninjas.amazon.com/s3files/s3get.cgi/lloyds-bank-roman-numerals-sm.jpg)

[From [S3InternalFiles](https://w.amazon.com/bin/view/S3InternalFiles), [Change](https://improvement-ninjas.amazon.com/s3files/s3put.cgi?s3key=lloyds-bank-roman-numerals-sm.jpg)]

Lloyds Bank near LHR15

Paco Hope (pacohope@) added:

In the real world, Roman numerals are not used as strictly as they are presented here. For example, Lloyds bank building in Leadenhall, near [LHR15](https://inside.amazon.com/en/MyWorkplace/AmazonBuildings/GlobalCorporateOffices/EMEA/UnitedKingdom/LHR15/Pages/default.aspx) uses "MDCCCCXXV". So perhaps the rules are not quite **set in stone**.

**The Date Problem**

**Description:**

A purposely ill defined problem, that is much more complex than it looks on the surface. The problem is designed to see if the candidate asks for clarification of requirements, can come up with a reasonable approach to a reasonably complex problem s/he has never thought about before, can communicate that approach, can come up with the reasonable test cases and execute the logic s/he has designed.

**Focal Categories:**

Culture Fit I (smart, asks questions, changes directions easily), Computer Science Concepts (programming principles, can think and develop algorithms), Quality Focused Design Skills (can explore in a minor way) and Coding Skills (more of a can this person write pseudo code).

**Phrasing:**

Write an algorithm that will take two dates and tell you if they are more than a month apart, less than a month apart or exactly a month apart.

**Ranked Solutions:**

This is not a question where the answer is the important thing. These are however, several kinds of answers you typically see:

* Uses provided library functions (not too good - no library does it, tends to mean we have an assembler not an engineer). This kind of answer typically means that the candidate has not asked what a date is, or s/he doesn't really understand what library date routines do.
* Convert to Julian notation, subtract the days and compare to 30. A failed answer - a month is not 30 days.
* Build lots of 'if' statements comparing days, months and years. A clumsy but reasonable approach. When you see this you can probe for 'how would you test this' and work the test case through the provided pseudo code.
* Convert to Julian notation, get the number of days difference and then figure out how many days are in the current month. This is one of the better approaches you are likely to see. You can probe on how to compute the number of days in February if you see this one.

|  |  |
| --- | --- |
|  | Comparing the difference with the # of days in current month is not sufficient. For example: 01-29-17 vs 03-01-17 is exactly 31 days apart, but is more than a month apart. One should compare if the days part match too. - [**miroslar@**](https://phonetool.amazon.com/users/miroslar) |

* A mathematical approach, (only one candidate has [EVER](https://w.amazon.com/bin/view/EVER) gotten this answer). Create a 'Julian Month' notation (number of months since some starting date) multiply by 100, add the date portion, subtract the two dates and compare the difference to 100.

**Hints:**

* You are letting the candidate proceed in their own path, unless they get flustered. Sometimes it helps to allow them to ignore the year for awhile.
* Reassure the candidate that s/he is writing pseudo code, not Java or C.
* Allow any convenient date format.
* Be in-exact in the definition of what a month is. Use examples such as 'January 15 to February 15 is exactly one month. February 15 to March 15 is exactly one month,' etc.

|  |  |
| --- | --- |
|  | Interesting - this invites the clever candidate to compare the day of the month, and if they're the same see if the months are adjacent. If they are, dates are a month apart. Similarly, adjacent months lets you do a simple numeric comparison on the day to determine less or greater than. However, without that how do we define what 'exactly 1 month apart' means? - [**paulbohm@**](https://phonetool.amazon.com/users/paulbohm) |

* Some design patterns are better than others. If s/he goes down a slow path, you may want to turn the problem by ignoring years.

**Indicators:**

* You want to see if s/he refines the definition. An early question should be 'What is a month? How is a date represented,' etc.
* If the candidate uses the 'if' solution, you can let them plow into it, then start asking boundary conditions which they have missed. See how they react to errors being pointed out.
* Ask the candidate to define some test cases. Look for edge conditions, bad inputs, etc.
* Run the test cases through the paper algorithm. See if the candidate can act as a human computer executing the pseudo code s/he has produced.

by John Chenault

**Shortest Answer:**

// returns: <1 if less than one month apart

// 1 if exactly one month apart

// >1 if more than one month apart

float month\_diff(int y1, int m1, int d1, int y2, int m2, int d2) {

return fabs((y1\*12 + m1 + d1/32.0) - (y2\*12 + m2 + d2/32.0));

}

olivwong@: Does this function claim that March 31st is exactly one month away from May 1st? What do you think an end user would expect it to return?

Java Implementation: <https://w.amazon.com/index.php/User:Arman/Interview#27._Date_Difference>

\* another way is to total number of days before dt1 from i.e., total days from 00/00/0000 to dt1, then count total number of days before dt2. Finally return the difference between two counts.

Let the given two dates be "1-Feb-2000" and "1-Feb-2004"

dt1 = {1, 2, 2000};

dt2 = {1, 2, 2004};

Count number of days before dt1. Let this count be n1.

Every leap year adds one extra day (29 Feb) to total days.

n1 = 2000\*365 + 31 + 1 + Number of leap years

Count of [http://geeksquiz.com/c-program-check-given-year-leap-year/ leap years] for a date 'd/m/y' can be calculated

using following formula:

Number leap years

= y/4 - y/100 + y/400 if m > 2

= (y-1)/4 - (y-1)/100 + (y-1)/400 if m <= 2

All above divisions must be done using integer arithmetic

so that the remainder is ignored.

For 01/01/2000, leap year count is 1999/4 - 1999/100

+ 1999/400 which is 499 - 19 + 4 = 484

Therefore n1 is 2000\*365 + 31 + 1 + 484

Similarly, count number of days before dt2. Let this

count be n2.

Finally return n2-n1

There are a lot of technical areas you can probe with this question, though it's primarily a design question.

**Prelude**

I'll first ask a candidate to define the term "garbage collection" to get us on the same page. If they have no idea what it is, no big deal, you can still have them work on the problem. This is roughly what the candidate should understand about garbage collection (from the [GC](https://w.amazon.com/bin/view/GC) [FAQ](https://w.amazon.com/bin/view/FAQ) [[1](http://iecc.com/gclist/GC-faq.html)]):

A [GarbageCollector](https://w.amazon.com/bin/view/GarbageCollector) is a part of a language's runtime system, or an add-on library, perhaps assisted by the compiler, the hardware, the [OS](https://w.amazon.com/bin/view/OS), or any combination of the three, that automatically determines what memory a program is no longer using, and recycles it for other use. It is also known as ``automatic storage (or memory) reclamation*.*

**Problem statement**

I pose the main part of the problem as a set of requirements and constraints; the candidate must come up with a design that meets the criteria. "Let's say we have a Java virtual machine that does everything but collect garbage memory. Your goal is to design an accurate garbage collector for this [JVM](https://w.amazon.com/bin/view/JVM) - that is, when the collector runs, it must reclaim any memory that can be reclaimed; a conservative solution that lets memory leak is not sufficient."

If you'd rather pose this in the context of a C++ runtime environment, feel free. Assuming we're implementing a [GC](https://w.amazon.com/bin/view/GC) for a [JVM](https://w.amazon.com/bin/view/JVM) tends to make some parts of the problem easier, though.

My typical problem constraints:

* there can be any number of "mutator" threads in the system
  + mutator threads modify the stack and heap
* there is one garbage collector thread
* when the garbage collector thread runs, all of the mutator threads are guaranteed to be at a "safe point"
  + safe points are the call sites, dynamic link sites, thread yield sites, possible exception-throw sites, and allocation request sites [[2](http://www-4.ibm.com/software/developer/library/jalapeno/)]
  + this sort of [GC](https://w.amazon.com/bin/view/GC) is known as a "stop-the-world" collector
* there is one shared heap
* you have access to every stack, the heap, and all object run-time type information ([RTTI](https://w.amazon.com/bin/view/RTTI))

The garbage collection entry point is some method (like System.gc() in Java) that is called by the runtime environment at a safe point. The candidate must determine what additional data structures they'll need, the [GC](https://w.amazon.com/bin/view/GC) algorithm, and the efficiency of their algorithm.

**Possible problem solutions**

1. **Mark-and-sweep collector** - good solution

*Approach*: to find all of the unused memory on the heap, you start with all of the "roots" of the system, trace all paths from each root, mark the used objects, and free the objects that were never marked during your graph traversal.

*Starting point*: in Java, roots are the live object references on the stack. (Bonus point if the candidate figures out that class variables, a.k.a. "static variables", are also roots.)

*Data structures*: you will need to record all heap allocations in some table. You will also need a corresponding bit vector of live/dead (used/unused) objects. If it is not easy to get type information for each entry on the stack (you determine if you want to make the candidate suffer on this point or not), then you will need another bit vector per stack, holding object/non-object flags for every stack entry. (Without type information, a type like an int could look just like an object reference, since most types in Java are 32 bits. If your int contained the same value as a valid memory reference, then you could end up with a pointer-aliasing problem, in which the piece of memory appears to be referenced. This causes your collector to be too conservative.)

*Performance*: The algorithm is a standard graph traversal algorithm (either breadth-first or depth-first), in which traversal is pruned when the "used" bit field is detected as already being set.

2. **Reference counting** - poor solution

You cannot reclaim cyclic garbage (like a circularly-linked list) with a pure reference-counting scheme. This does not meet our requirement to design an accurate collector.

**Advanced topics, extensions**

If you really want to test the candidate, try these problem extensions:

* how do you compact the heap after freeing memory?
  + the candidate should address how object reference implementations are affected - are object references direct or indirect? Direct references make for faster program runtime, but heap compaction becomes more complex. Indirect references slow down runtime, but can make heap compaction an O(n) process (where n is the number of objects on the heap).
  + copying collectors have a "from space" and "to space" - what's left in the "from space" is garbage, and everything in the "to space" is live. The role of the spaces is reversed with each collection phase.
* If you had multiple heap spaces, how could they improve the performance of your algorithm?
  + generational garbage collectors partition objects into old and new spaces. A simple version would put all new objects into the new space, and would move objects that survived *n* number of [GC](https://w.amazon.com/bin/view/GC) runs into the old space. By doing that, you can modify your [GC](https://w.amazon.com/bin/view/GC) algorithm to check the new space more frequently than the old space; among other things, this prunes your search space, and increases your locality of reference.
* let the mutator threads run at the same time as the [GC](https://w.amazon.com/bin/view/GC) thread (that is, design an "incremental" garbage collector, rather than a stop-the-world collector)
* how would you test your garbage collector implementation? How do you know that it is accurate?
* how do you deal with C/C++ programs that have references to objects in the [JVM](https://w.amazon.com/bin/view/JVM) heap (acquired via [JNI](https://w.amazon.com/bin/view/JNI))?

Relatively easy coding question for an [SDE](https://w.amazon.com/bin/view/SDE) candidate (good for college recruits).

**Problem statement**: on the whiteboard, write a C function that will delete a node from a circularly-linked, singly-linked list.

**Issues**: The candidate will need to define the node structure. They can assume that there is some global "list" pointer that points to the head of the list. Their solution must handle these special cases:

* list is empty
* list has only one element
* first element in list is the one to be deleted
* last element in a list is the one to be deleted (or is it?)
* the element isn't in the list
* the element is in the list more than once

If they don't get all of these cases up front, you can give small hints. Candidates who need more help are probably not going to be good developers here.

They also should be able to explain the complexity of the task in O(n) terms.

Relatively easy coding question for an [SDE](https://w.amazon.com/bin/view/SDE) candidate (good for college recruits).

**Problem statement**: on the whiteboard, write a C function that takes two strings, and returns the index of the first occurrence of one string in the other.

**Issues**:

* if they ask for the return type, ask them what they think it should be. An int is fine.
* if they ask what to do if the string isn't found, again, ask them what they think it should do. Returning -1 is okay for this example.
* they can't use C string functions like strcmp, strncmp, and so forth. If you want to give them strlen, feel free.
* their solution should cover all of these cases:
  + string not found
  + either string is null
  + either string is larger than the other
  + search string "aaba" in "aaabacde" - this is the scenario most candidates miss, because their loop advances too far when there is a mismatch.
  + [KMP algorithm](http://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm)
* ask the candidate how they would test their solution. Given that they cannot try every possible set of test cases, what intelligent approach can they take? Make sure their test exercises each part of their code - if they don't get full code coverage on such a simple function, they'll likely have problems testing larger cases.
* their function signature should be something like this:

int find\_string(const char \*needle, const char \*haystack) { ... }

This is a conceptually simple problem that can be solved with a fairly straightforward recursive algorithm. It is a good test of a programmer's ability to recognize and solve problems that have recurrences. There is also an efficient iterative solution to the problem that requires a little bit of thinking outside the box. The problem is this:

You are given an array of characters in some language. How would you go about printing all the subsets of characters from this array? For example, if the array you are given is [A,B,C], you would print:

A B C [AB](https://w.amazon.com/bin/view/AB) [AC](https://w.amazon.com/bin/view/AC) [BC](https://w.amazon.com/bin/view/BC) [ABC](https://w.amazon.com/bin/view/ABC)

Note that order doesn't matter, so that the subsets [A,B] and [B,A] are the same (this makes the problem much easier).

The recursive solution to this problem is as follows:

void printSubsets( char\* set )  
 {  
     char subset = char[256];  
     subset[0] = '\0';  
   
     printSubsetsR( set, subset );  
 }  
   
 void printSubsetsR( char\* set, char\* subset )  
 {  
     int N = strlen( set );  
   
     **for**( int i=0; i<N; i++ )  
     {  
         char new\_subset[256];  
   
         strcpy( new\_subset, subset );  
         strncat( new\_subset, set+i, 1 );  
         printf( "%s**\n**", new\_subset );  
   
         printSubsetsR( set+i+1, new\_subset );  
     }  
 }

The recursive solution in a functional language (Clojure):

(**defn**print-subsets-aux [set subset]  
  (reduce (**fn**[ind \_]  
            (**let**[new-subset (str subset (.charAt set ind))]  
              (println new-subset)  
              (print-subsets-aux (.substring set (inc ind)) new-subset))  
            (inc ind))  
          0 set))  
  
(**defn**print-subsets [set]  
  (print-subsets-aux set nil))

The iterative solution involves noticing that each element of the original set is either in or out of each subset. So, for each possible subset, every element can be thought of as a bit (on or off). There are 2^N possible subsets, which correspond to an N bit binary number. This leads to the following code:

void printSubsets( char\* set )  
 {  
     int N = strlen( set );  
   
     **for**( int i=0; i<pow(2,N); i++ )  
     {  
         **for**( int j=0; j<N; j++ )  
             **if**( i & (0x1 << j) )  
                 printf( "%c", set[j] );  
   
         printf( "**\n**" );  
     }  
 }

Iterative solution in a functional language (Clojure):

(**defn**print-subsets [set]  
  (**let**[n (.length set)]  
    (dotimes [i (Math/pow 2 n)]  
      (dotimes [j n]  
        (when (> (bit-and i (bit-shift-left 0x1 j)) 0)  
          (print (.charAt set j))))  
      (print "\n"))))

Java Implementation: [User:Arman/Interview#7.\_Subsets](https://w.amazon.com/bin/view/User:Arman/Interview#7._Subsets)

Python implementation: [User:Mayali/Interviews#StringSubsets](https://w.amazon.com/bin/view/User:Mayali/Interviews#StringSubsets)

Code up a simple class in the language of your choice to represent a deck of cards with operations to shuffle the deck and to deal one card.

Ordering Services requests this from candidates as a coding assignment after the first phone screen. The principle information garnered is "Can the candidate solve a simple problem in a simple, straight-forward way?" Its mostly a negative test, to weed out candidates that would otherwise make it through the phone screen process but are not actually decent coders. There are more of these people than one might at first think. The majority of people fail this test, and many submissions are quite bizarre.

It is important to keep in mind that if a candidate makes a decent submission, it only weights to a minor degree in their favor, since the problem is so simple. A good submission only indicates that we should continue with the regular interview process.

**Wording**

The question is purposely left somewhat general and vague to see where a candidate goes with it.

**Submission Quality**

A good submission is one which is relatively short and simple, and follows common programming practices. What really stands out are submissions that are not short, or are not simple, or that contains code where one would say "why the heck did they do that?" Ordering services has 2-3 people critique each submission, listing observations of flaws and ranking them from 1 (astonishingly bad) to 5 (a little bad). If a submission is particularly nice for some reason, comments are made on that as well.

Below are a number of typical flaws to serve as examples. But after reviewing submissions for a while, one quickly learns that there is no limit to the number of "interesting" variations there are for this problem.

**Deals Same Card Repeatedly**

Repeated calls to the deal() method always return the same card, i.e. a card is not removed from the deck when it is dealt.

**Init without using For-loop**

Does not use for-loop to initialize cards in the deck, i.e. something like

void Deck::init()

{

m\_cards[SPADES][ACE] = Card(SPADES, ACE );

m\_cards[SPADES][TWO] = Card(SPADES, TWO );

m\_cards[SPADES][THREE] = Card(SPADES, THREE);

...

}

**Implements Own Linked List**

Implements own linked list class to store cards in the deck, as opposed to using [STL](https://w.amazon.com/bin/view/STL) vector or list, or simply using an array.

**Deck of Integers**

Doesn't create a card class. Just creates a deck of 52 integers with no reference to card suit or rank.

class Deck

{

...

int deal();

private:

int m\_cards[52];

};

**Over Qualifies Names**

Redundantly encodes class name into member names.

class Card

{

Suit m\_cardSuit;

Rank m\_cardRank;

};

class Deck

{

...

void shuffleDeck();

};

**Sort Key in Card**

Adds an integer member to card class to store random keys in and then use to shuffle deck.

class Card

{

Suit m\_suit;

Rank m\_rank;

int m\_key;

};

**Strings for Suit and Rank**

Card class uses strings to store suit and rank. Works for this problem, but not a very general notion of what a card is. Would not allow card to be easily used in any card game where comparison of rank was required.

class Card

{

String suit;

String rank;

};

**Extra Features**

Included a whole mess of methods and behavior not asked for.

class Deck

{

public:

Card dealTop ();

Card dealBottom();

vector<Card> dealTop (int n);

vector<Card> dealBottom(int n);

void addTop (Card c);

void addBottom(Card c);

void cut();

};

**Player & Game**

Creates player, game, or other similar classes. The problem statement doesn't request these.

**Unnecessary Copy Constructor or Assignment**

Hand writes a copy constructor or assignment operator in the card class even though they are not needed.

class Card

{

Card();

Card(Suit s, Rank r);

Card(const Card& c);

Card& operator=(const Card& c);

...

};

**Memory Leaks**

Memory leaks exist in the code. For example, deck uses dynamic memory to allocate cards, but fails to provide a destructor.

class Deck

{

...

Card\* m\_cards[52];

};

**Deck Copy Fails**

Uses dynamic memory for cards, but fails to provide a copy constructor and assignment operator for Deck.

class Deck

{

...

Card\* m\_cards[52];

};

**Private Copy Constructor & Assignment**

Make copy constructor and/or assignment private for Deck thus preventing clients from copying a deck if they want to. Worse if they do just one and not the other.

class Deck

{

private:

Deck(const Deck& d);

Deck& operator=(const Deck& d);

};

As a variation (maybe for [SDE](https://w.amazon.com/bin/view/SDE)-I or entry-level candidates) it might be better to ask them to build a Card class (if in Java), then introduce a Deck class. I asked this question to a few candidates in phone screens and it went a lot smoother when I started bottom-up.

Design the controls for a modern four-way traffic intersection with all the works: ~~weight~~ vehicle sensors, pedestrian lights & buttons, etc.

This is an interesting problem that tests the candidate's ability to think in terms of distributed systems, interrupt-driven (or event-driven) programming, state machines, and scheduling techniques.

<http://auto.howstuffworks.com/car-driving-safety/safety-regulatory-devices/question234.htm>

<http://www.ccs.neu.edu/home/vkp/Papers/Traffic-sigcse98.pdf>

You have a singly linked list. How do you find the 3rd element from the end?

You or the candidate should clarify what is meant by "3rd from the end".

**Good solutions**

* Make one pass to count the number of elements.
* Make a second pass that pulls out the length - 3 element.

-or-

* Keep a pointer that's 3 behind the current pointer

The best solutions will address when the list is shorter than 3 - how is that case identified and communicated back to the caller?

**Best solution**

* One pass: (car (last n 3))

(defun cl:last (list **&optional** (n 1))  
    (**declare** (values t))  
    (**declare** (**type** list list))  
    (**declare** (**type** sequence-index n))  
  (**let** ((runner list)  
        (result list)  
        (i 0) )  
    (loop  
      (when (atom runner) (return result))  
      (when (>= i n) (pop result))  
      (pop runner)  
      (incf i) ) ) )

**Bad solution**

* Keep last, second to last, and third to last pointers

If they give you the bad solution, ask them what they would do if you wanted the 50th from the last? Hopefully they'll be able to generalize the solution.

I've found this to be a good phone interview question. cade@ asked it during my initial phone screen, so props to him.

Note: this question is discussed in Programming Interviews Exposed: Secrets to Landing Your Next Job ([www.amazon.com/exec/obidos/ASIN/04713...](http://www.amazon.com/exec/obidos/ASIN/0471383562/)), so there's a chance candidates will have seen it before.

**Extensions / Variations**

**M from the End**

Ask to find the element M items from the end (where M=0 -> last element)

**Discussion**

**Not much reason to judge a solution good or bad**

It's still a decent question to see how well the candidate thinks, and how he can explore the solution space.

But after asking this question many times, I've grown not to like it so much, because there is very little practical difference (in terms of performance) between the quality of the two "good" solutions. In fact, even the "bad" solution might outperform the good solutions under certain circumstances.

I used to prefer that candidates come up with 1-loop "good" solution over the 2-pass "good" solution. But both involve the same number of link traversals, so in terms of machine instructions executed, there's not so much difference. The 1-loop solution might perform better when the effect of memory caches are considered, as the nodes are more likely to be in cache when the second pointer visits them.

If you're judging the solution by performance, and not elegance, then for the limited case of M=3 (or other small number), in the "bad" solution there's only one actual access of the node & its next pointer per iteration. Keeping track of the last 3 pointers is probably going to be optimized to use the registers, so that will be very fast.

Going even farther, if access to each node is relatively expensive (maybe they're not even in memory, but have to be paged in or are coming from a remote service), then this could be a big win.

**Algorithm Comparison**

As is pointed out, the 1-loop (trailing reference) v. 2-loop (length based) solutions are nearly identical in terms of performance. It is possible to turned this into a richer question by getting the candidate to a point where they've identified both algorithms. Sometimes it is necessary have to help them get to both algorithms. Ideally they can implement both but that may take too much time. Then get the candidate to decide which algorithm is best to implement. Some candidates insist the 1-loop implementation is better despite the Big-O analysis. If the candidate realizes that both are similar in terms of performance they can struggle to find other ways to evaluate the code. A good hint is asking the candidate how they'd review the code if it was written by someone else. This gets them thinking along different lines. The 2-loop/length based solution is refactor-able into separate functions, this makes the code more reusable, improves readability and testability.

**Public Solutions for this exercise**

This is from "Programming Interviews Exposed 3rd Edition", be suspicious if you get a very similar answer:

ListElement \*findMToLastElement( ListElement \*head, int m ){  
    ListElement \*current, \*mBehind;  
    int i;  
    **if** (!head)  
        **return** NULL;  
    /\* Advance current m elements from beginning,  
    \* checking for the end of the list  
    \*/  
    current = head;  
    **for**( i = 0; i < m; i++ ) {  
        **if**( current->next ){  
            current = current->next;  
        } **else** {  
            **return** NULL;   
        }  
    }  
    /\* Start mBehind at beginning and advance pointers  
     \* together until current hits last element  
     \*/  
    mBehind = head;  
    **while**( current->next ){  
        current = current->next;  
        mBehind = mBehind->next;  
    }  
    /\* mBehind now points to the element we were  
     \* searching for, so return it   
     \*/  
    **return** mBehind;  
}

**implement strstr()** is a fairly well-rounded question. I like to ask it as a warmup at the beginning of onsite interviews, as it gets the candidate used to writing code on the board, and it can be a pretty good barometer of their coding ability.

It's *always* a good idea to see actual code, even if you're just testing on design. Some people can talk through design pretty well, but fail miserably at basic code-writing. If you give this question during a phone screen, have them read the code to you verbatim.

I usually phrase it as "please write me a function to find the first occurrence of one string in another string". I sometimes add the restriction of doing it without using the functions in string.h

I give no more than 10 to 15 minutes for this question; if somebody is having a hard time with the problem in that timeframe, they're probably not a good candidate.

Once the basic solution has been found, I ask the candidates how they would optimize the solution.

**Barometric pressure:**

If the candidate can't produce a working solution, they're a "no hire". Period. A first or second year college student should be able to produce reasonable code to do this. One of the strangest failure cases for this question that I'd encountered actually wanted to solve it recursively! *Peanut gallery: I don't think a recursive solution is a failure for a first attempt, as long as they understand the practical issues with recursion and how to transform it into an iterative solution.*

**Your** [**SDE**](https://w.amazon.com/bin/view/SDE) **I candidate will**

* give the baseline solution - nested for loops that compare each character of the needle string against the haystack string, and advancing one character in the haystack string if a match was not found.
* may ask about return types, and what to return if a match isn't found.
* may ask to use strlen()
* If you gave them strlen, they shouldn't call it excessively (ie, in the test section of the for loop).
* may walk through the entire haystack string, rather than just the first strlen(haystack) - strlen(needle) + 1 characters.

**Your** [**SDE**](https://w.amazon.com/bin/view/SDE)[**II**](https://w.amazon.com/bin/view/II) **candidate will:**

* recognize this as strstr.
* set the pointers as const char \* in the fn signature.
* check to make sure the inputs are not null pointers.
* (dawalker@) I disagree that an [SDE](https://w.amazon.com/bin/view/SDE) [II](https://w.amazon.com/bin/view/II) or stronger will check if the inputs are [NULL](https://w.amazon.com/bin/view/NULL). Libc's strstr() doesn't do this - it is implicit in the calling contract that needle and haystack are valid strings. It's one of subtle differences between C and C++.
* use pointer arithmetic rather than length + array indicies.
* may put constants on the LHS of comparisons.
* be able to walk through the strings by checking for the null byte rather than using strlen.
* be able to give a least some good optimizations, perhaps after coding the general solution.

A decent baseline solution might be:

int mystrstr( const char \*needle, const char \*haystack ) {

const char \*cur\_p = haystack;

if ( 0 == needle || 0 == haystack ) {

return -1;

}

while ( \*cur\_p ) {

int i;

const char \*p, \*q;

for ( p = needle, q = cur\_p;

0 != \*p;

++p, ++q ) {

if ( \*p != \*q )

break;

}

if ( 0 == \*p ) {

return (int)(cur\_p - haystack);

}

++cur\_p;

}

return -1;

}

There are several optimizations that are possible here:

* Recognize that you can eliminate the need to examine every substring by adding up the values of the characters in the needle string, and keeping a running total of the current strlen(needle) characters from the haystack string - if the values are different, the substring obviously doesn't match. The candidate should not create a separate character-to-value mapping, and they should understand that this technique can give false positives (aaab and baaa, or abbc and bbbb).
* Checking whether needle is longer than haystack is [NOT](https://w.amazon.com/bin/view/NOT) really a good optimization; even the baseline solution will only need to walk through the needle string once before returning false on comparing against the [NULL](https://w.amazon.com/bin/view/NULL) bye in the haystack string. Calling strlen will walk through both strings completely; you only save strlen(haystack) comparisons.

**Your** [**SDE**](https://w.amazon.com/bin/view/SDE)[**III**](https://w.amazon.com/bin/view/III) **candidate will:**

* think through the optimizations before writing code.
* They may also come up with a Monte Carlo type solution for generating substring values in such a manner as to eliminate false positives, using prime numbers and a polynomial equation rather than a simple sum. (I have yet to actually see this produced in an interview, but I still have my hopes.)
* (tonyw) Suggest other ways to advance though the haystack more rapidly. I'm still waiting for someone to implement a Boyer-Moore solution in an interview.

**Most Efficient Algorithm**

[Yoinoue](https://w.amazon.com/bin/view/User:Yoinoue) 08:23, 9 December 2009 (UTC) This isn't easy to answer question for fast algorithm. :-)

See collection of algorithm in [ASIN: 0954300645](http://www.amazon.com/gp/product/0954300645) Handbook of Exact String Matching Algorithms (Paperback)

As far as I know Search algorithm is the most efficient for large alphabets and small pattern [Lecroq95]. It is faster than Boyer-Moore, Turbo-BM, BM-Horspool.

Features from [Charras04]:

* Simplification of the Boyer-Moore algorithm;
* Uses only the bad-character shift;
* Easy to implement;
* Pre-processing phase in O(m+s) time and O(s) space complexity;
* Searching phase in O(mn) time complexity;
* Very fast in practice for short patterns and large alphabets

where m is length of pattern, n is length of test, s = size of alphabet.

**Reference**

* [Charras04] Christian Charras and Thierry Lecroq, Handbook of Exact String Matching Algorithms, King's College London Publications, 2004, <http://www-igm.univ-mlv.fr/~lecroq/string/>, [ASIN: 0954300645](http://www.amazon.com/gp/product/0954300645)
* [Sunday90] Sunday D.M., 1990, A very fast substring search algorithm, Communications of the ACM . 33(8):132-142.
* [Lecroq95] Lecroq, T., 1995, Experimental results on string matching algorithms, Software - Practice & Experience 25(7):727-765, <http://www-igm.univ-mlv.fr/~lecroq/articles/spe95.pdf>
* [Michailidis99]P.D. Michailidis and K.G. Margaritis, On-line String Matching Algorithms: Survey and Experimental Results, International Journal of Computer Mathematics, 76(4), pp. 411-434, 2001, <http://macedonia.uom.gr/~panosm/pdfs/jpaper001.pdf>

## Intro

This question could be used as a warmup question or a weed out. It assumes that the person being interviewed has some experience with the C standard library. This is derived from an actual bug in the Amazon.com codebase (authored by an intern).

## Original (Buggy) Code

To start, put the following code on the board (or tell them over the phone).

1 std::string bytesToHex(char \* bytes, size\_t size)

2 {

3 std::string result;

4 for (int i=0; i<size; i++) {

5 char hexByte[2];

6 sprintf(hexByte, "%02x", bytes[i]);

7 result.append(hexByte);

8 }

9 return result;

10 }

## Questions

Then ask them the following questions.

### What is wrong with this code?

If they're good they'll figure it out right here. The problem is that the 'hexByte' character array is one character too short. When sprintf writes the hex digit to 'hexByte', it will write two characters of hex followed by a null string terminator, for a total of three characters written to 'hexByte'. Since 'i' is the most recent variable, it will probably be the one that gets trampled by the buffer overflow, which means that 'i' keeps getting reset to 0 and thus the loop becomes infinite.

### Does this function ever return?

The answer is no, or probably not (depends on the compiler). Ask this only if they don't immediately understand what's going on with the code. If you need to, you can tell them that you ran the code through a debugger and put a breakpoint at line 7. You printed out the variable 'i' each time and found that it was always zero.

### How would you fix this function?

Once they understand what's wrong they should be able to fix the code by changing the size of hexByte to 3. If they're good, they may also decide to switch from sprintf to snprintf and do error checking on the return value (snprintf should always return 2 for this usage). You may need to lead them a little to realize the need to switch to snprintf.

### Why should you never use sprintf, and what can be used to replace it?

If they've had any training in writing secure code they should know that sprintf is evil given the potential for buffer overflows. The bug would probably have been detected much easier if snprintf was used in the first place, and the return value was checked for correctness.

## Final (Fixed) Code

Following is a fully bugfixed version of the bytesToHex function.

1 std::string bytesToHex(char \* bytes, size\_t size)

2 {

3 std::string result;

4 for (int i=0; i<size; i++) {

5 char hexByte[3];

6 if (2 != snprintf(hexByte, 3, "%02x", bytes[i])) {

+ fprintf(stderr, "failed to write hex byte '%02x' to string.", bytes[i]);

+ abort();

+ }

7 result.append(hexByte);

8 }

9 return result;

10 }

**Warning**

There are better interview questions for probing "Data Structure and Algorithms" competency.

This interview question is poor because:

1. No matter the candidates' level, this question requires the candidate to demonstrate both problem solving AND *knowledge of* algorithms. You risk not getting data points on data structure. This failure can play out a number of ways. In the worst case they don't get past the naive solution to have anything interesting to work with. In the best case they "breeze through" it by raw knowledge of algorithms. In any case it doesn't guarantee the candidate has demonstrated Data Structure competency.
2. For SDE-I level candidates, especially newgrads, this problem requires the *interviewer* to be on top of everything, communicating very well, breaking down the problem into small chunks. It's a long problem, and nervous candidates will miss some things.

It's probably better to find another problem.

**Problem**

There is a common type of word puzzle (called [word ladders](http://en.wikipedia.org/wiki/Word_ladder) or doublets) where you are given two English words of the same length, say, "HEAD" and "TAIL". The puzzle is to come up with a sequence of valid English words, starting with "HEAD", and ending with "TAIL", such that each word is formed by changing a single letter of the previous word. Create an algorithm to automatically solve such puzzles.

Example (altered letters capitalized):

HEAD

heaL

Teal

teLl

tAll

taIl

**Observations / Variations**

* Not every word in a solution necessarily uses the corresponding letters from either the beginning or ending word (in the example above, 'L' in the 3rd position of "TELL" and "TALL" doesn't appear in HEAD or TAIL. Some solutions may *require* using letters that don't appear in either the starting or ending words
* You may choose to require that the algorithm return the shortest possible solution, or not.
  + If you require it, then you may be putting the candidate at a disadvantage, because it requires them to be able to identify some very particular techniques that lend themselves to shortest path solutions.
  + I usually don't require it, because I want to give people enough room to explore the solution space on their own.
* A candidate who happens to be familiar with basic graph theory will tend to be able to identify the breadth & depth-first approaches sooner than other candidates. You can go on to ask them to code the algorithm, to see if they understand it well enough to put it into code.

**Solutions**

All solutions will require some way to determine if a word is an english word, so allow them some sort of dictionary. If they do well enough at the basic algorithm, there is room for optimizing it with a specialized dictionary, so I don't usually ask about the details of the dictionary straight away.

Basically, this is a graph problem, and it's nice if the candidate identifies it as such. A full solution involves generating the graph (explicitly or implicitly), plus a path searching algorithm.

**Depth First**

Virtually all candidates who aren't well versed in graph theory take this approach, probably because it most closely models how they themselves would solve this kind of problem. Elements of a correct solution include:

* How to generate the next set of candidate words
  + Either generating them on the fly by correctly looping through positions & letters in the alphabet.
  + They might pre-process the dictionary into a data structure that returns all 1-letter adjacent words from any given word. This is especially helpful if the algorithm is expected to be run many times in succession, as it eliminates the overhead of the generate & test.
* How to keep track of the sequence of words and return the solution to the caller
  + They might keep an explicit stack/list somewhere, and modify it is they traverse the graph
  + Or just keep the current search path on the call stack
  + Or maintain a set of back-pointers from each word they visit, to the previous word that they visited. When the final word is reached, you
* Avoid infinite loops, which usually means keeping track of at least every word in the current search path, and skipping those that are encountered again.
* Knowing when to stop descending down a branch (i.e., they've found a solution, or there isn't one).

Elements that make a solution better are:

* Not only avoiding re-use of words that are already in the current search path, but any word that has ever been visited during the search.
* If they're trying to find the shortest path, then they need to keep track of the "best answer so far", and stop going down a branch if it has become longer than that best answer.
  + If they're also keeping track of already-visited words, then they'll need to augment that with information about how many steps it took to visit that word, and continue if the current solution has arrived at that word in fewer steps.
* Avoiding useless substitutions: Changing the same position back-to-back in successive words. Such substitutions make a solution unnecessarily long. E.g., HEAD -> BEAD -> LEAD could just be HEAD -> LEAD

**Breadth First**

This could also be called a degenerate form of [Dijkstra's Algorithm](http://en.wikipedia.org/wiki/Dijkstras_algorithm), where the edge weights are the same value.

A thorough answer should talk about:

* dequeueing the next candidate word
* generating all the next possibilities
* enqueueing those on the end of the queue.
* How to track the sequences of words leading to each possibility
* Avoiding re-use of words (same as in depth-first)

**A\***

[https://en.wikipedia.org/wiki/A\*\_search\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)

A\* search would be suitable too for finding the shortest path. The heuristic function would be based on how many letters are incorrect, which is a lower bound on the number of remaining steps.

**Bidirectional Breadth First**

Same as BFS, but we start searching from both ends at the same time and stop as soon as we find common word in both search trees. Improvement comes from the fact that two circles with radius r have smaller total area than 1 big circle with radius 2r.

Complexity of naive BFS for two words N connections apart in the network with M connections per word on average is O(M^N), while complexity of bidirectional search will be O(M^(N/2)).

This is an interview question that's been serving me well lately. It covers algorithm design, data structure trade-offs, run time analysis, and coding skill. There's also a little ambiguity to handle.

Start with a vector of N Jobs. Each Job has a type[ID](https://w.amazon.com/bin/view/ID), an arbitrary integer that can be used to associate Jobs with each other. Let's assume this can be obtained from int Job::getType().

We would like to process all N Jobs using some technique, but for performance reasons (e.g. [CPU](https://w.amazon.com/bin/view/CPU) utilization) cannot process more than K jobs at a time.

We would like Jobs of the same type to be processed together (e.g. Cache Localization, etc)

Develop and implement a strategy to break the N jobs into batches of less than or equal to size K. Strive to keep jobs with the same type in the same batch. Also strive to achieve batch sizes that approach K.

@manuanan  
I enhanced this question to process jobs in order. The order is defined as increasing order of the arbitrary integer associated with the job

Also since I was asking this question for SDE II level, I also added another dimension of time to job and asked candidate to optimize for time

Reversing a singly-linked list is a good phone screen question. You should write code to do this before asking the question!

Regardless of whether the candidate starts with a recursive or iterative solution, you should ask them to produce both.

The *basic recursive solution* will set the head list pointer to the value of a recursive call which defers evaluation of the return value until the last node in the list. The functions basically look like this:

Node \*Node::reverse\_rec( void \*prev ) {

Node \*retval = this;

if ( next != NULL ) {

retval = next->reverse\_rec( this );

}

next = (Node \*)prev;

return retval;

}

void List::reverse( void ) {

head = head->reverse\_rec( NULL );

}

Here is an alternative method that doesn't use a helper function:

Node \* reverse( Node \*n ) {

Node \*m;

if ( n == NULL || n->next == NULL )

return n;

else {

m = reverse( n->next );

n->next->next = n;

n->next = NULL;

}

return m;

}

The *basic iterative solution* will use 3 pointers (prev, cur, and next) and looks like this:

void List::reverse( void ) {

Node<C> \*prevp = NULL;

Node<C> \*curp = head;

while ( curp != NULL ) {

Node<C> \*nextp = curp->next;

curp->next = prevp;

prevp = curp;

curp = nextp;

}

head = prevp;

}

*Note that there may be some differences in the function signature depending on how they've set up the list. I generally put a "dummy" node on the front of my hand-written lists so that all inserts can be handled in a uniform manner (i.e., there's no special handling for adding to the head of the list). Of course, this is a memory vs. code simplicity tradeoff.*

I will generally have candidates write their algorithm down and talk me through it, then read the whole thing to me after they're done so that I can analyze the loop.

Unfortunately, this question can take a lot of time - plan for 10 to 15 minutes with sharp candidates and up to half an hour with the weaker ones. I generally don't allow candidates to continue beyond 30 minutes.

This is a fairly standard question, but from my experience, people are rarely prepared to do it both recursively and iteratively.

**Bonus points:**

* the candidate asks if the list is singly or doubly linked
* the candidate calls out the fact that *if ( ptr )* is equivalent to *if ( 0 != ptr )* or if( NULL != ptr)
* This is one of our [interview questions](https://w.amazon.com/bin/view/InterviewQuestions), first brought to Amazon by Chris Brown (cb@), as far as I know.
* Write a function that takes an array and size as a parameter. The array contains non-negative numbers. Every number in the array appears an even number of times, except for one number that appears an odd number of times. The function should return the number that appears an odd number of times.
* A useful follow-up question you should ask is, how would you test this code. There are lots of edge cases that should be tested for, and it's very revealing to see if the candidate can come up with a reasonable set of them.
* *Warning:* [this is Googleable](http://www.technicalinterviewquestions.net/2009/01/integer-array-odd-even-occurrences-find.html), so if your candidate immediately gives the XOR answer, you may want to move on to another question instead.

# Questions you might get

At this point, you should encourage the interviewee to ask clarifying questions. Most candidates fail to ask any, but you might get:

*Is a single-element array valid?* Yes.

*Does the array have to have any numbers that appear an even number of times?* No, although it would imply that there's only one number that appears in the array.

*Is a 0-element array valid?* No.

*Do I have to check for corrupt arrays?* Nope.

*Is there a bound on the numbers?* They're 32-bit unsigned integers. With modern hardware, 64-bit might be a better size if you want to make counting into an array not feasible.

*Is the array sorted?* No. (If they ask this question they're probably on the track of one of the solutions).

# Solutions

Next you should have the candidate walk through at least 2 different algorithms to solve the problem. Ideal candidates will think of several on their own and tell you about them, or at think of at least two with no more hints than, "Can you think of another way to do it?" There are 4 possible solutions, listed in roughly worst-to-best order.

## The O(N^2) solution

For each element of the array, walk through all the other elements of the array and increase a counter for every instance of the same number. If the counter is odd at the end of the array then return that number. Otherwise do the same thing for the next element.

A lot of candidates fail to see that, for each number, you have to look at *all* the other numbers in the array, not just the ones following the array. I.e., you have to do this (in [BLUB](https://w.amazon.com/bin/view/BLUB)):

unsigned findOdd(unsigned a[], unsigned size)

for (unsigned i = 0; i < size; ++i)

unsigned count = 0

for (unsigned j = 0; j < size; ++j)

if (a[i] == a[j])

count++

if (count % 2 == 1)

return a[i] // note: remember they don't need to handle corrupt arrays

There are some "optimizations" like turning matching numbers into -1's, etc, but basically this method sucks.

## The sorting solution

First sort the array (candidates do not need to write the sorting routine, but they should know that good sorting algorithms are O(n log n)). Then walk through the array in an O(n) pass to find the oddly-appearing number. Good candidates figure out the really short, easy way to do this, comparing 2 numbers at a time. Most candidates write a more complicated loop where they count the number of occurrences of the first number, see if that's odd, then count the number of the next number, etc.

The good solution is something like:

unsigned findOdd(unsigned a[], unsigned size)

qsort(a, size, ...mumble...)

for (unsigned i = 0; i < size - 2; i += 2)

if (a[i] != a[i + 1])

return a[i]

return a[size - 1]

Boundary conditions are a little tricky, make sure they think about them. The above works because the array is an odd size. I tell them not to worry about detecting corrupt arrays, it messes too much with this question's mojo.

Also see if you can steer them towards the simpler solution if they come up with the more complex one.

Bonus points for anyone who realizes the O(n) pass after the sort can actually be a sort of binary search-like O(log n) thing.

Bonus points for anyone who realizes that a O(n) search algorithm (like radix sort) could be used, since the input is limited to unsigned ints.

## The hashtable solution

Slightly faster, though potentially using more memory (sorting can use extra memory too, of course), is using a hashtable to store counts. I usually don't have them code this up, instead if they suggest this one I see if they can think of the sorting solution and then have them code that one.

If they use a hashtable, they should be able to realize that the most optimal thing is to use a hash set rather than a hash map, and just remove numbers from the hash set every 2nd time you see them. Most people think of using a hash map (where the values are counts of the number of times seen). The hash map has the downside of requiring an O(n) pass thru the keys of the map at the end, plus the code is slightly more complicated.

I usually don't have them write this code but here is roughly what it would look like:

unsigned findOdd(unsigned a[], unsigned size)

HashSet s

for (unsigned i = 0; i < size; ++i)

if (s.contains(a[i]) s.delete(a[i])

else s.insert(a[i])

return s.keys()->value()

## Use of relatively prime numbers (to uniquely identify input elements)

Another O(N) time, in-place algorithm (given that we have a set of relatively prime numbers):

int[] primes = {2,3,5,7,11,13,17,19,?,n}

int findOdd(int[] a)

int product = 1;

for (int i = 0; i < a.length; ++i)

if (product % primes[a[i]] == 0)

product /= primes[a[i]];

else

product \*= primes[a[i]];

int i = 0;

while (primes[a[i]] != product)

++i;

return a[i];

NOTE: this will work for any given set of relatively prime numbers (e.g., {2, 5, 9, 77, ?}).

FURTHER NOTE: the space require for the array of prime numbers is actually O(MAX\_INT\*log(MAX\_INTth prime)), since a[i] could be as large as MAX\_INT. This would require primes to have a[i] elements, each of which requires at most O(log(MAX\_INTth prime)) bits to store. Further, product can't be an int since an intermediate product could easily exceed the storage capacity of an integer, even if a[i] has a much lower bound.

## The XOR solution

I don't really expect anyone to get this, but the optimal solution is to just [XOR](https://w.amazon.com/bin/view/XOR) all the numbers in the array together.

unsigned findOdd(unsigned a[], unsigned size)

int oddNumOut = a[0]

for (unsigned i = 1; i < size; ++i)

oddNumOut ^= a[i]

return oddNumOut

*~~That won't catch zeros, which are explicitly allowed by the problem description ("non-negative"). You need a separate even/odd flag for the zeroes.~~*

*Yes, it does: if 0 is the odd-counted number, everything else cancels, leaving 0 in oddNumOut, which is correct.*

This won't catch certain error cases -- for example if more than one number is repeated an odd number of times (or none are).

**FYI:** A place a candidate might have seen this algorithm before is they've ever had to implement a [longitudinal parity check](http://en.wikipedia.org/wiki/Longitudinal_redundancy_check)

**FYI:** Many candidates with EE degrees should come up with this solution immediately - it is common to do this sort of thing in EE coursework/problems.

**Description**

This is a simple and powerful encoding algorithm, so I think it would make a good [SDE](https://w.amazon.com/bin/view/SDE) interview question for a candidate to write this out.

Assume we have a heap (keys are the characters/strings; values are their occurrences). In each step of the algorithm, the two rarest characters are looked at. Both get a suffix (one "0", the other "1"). They are joined together and will occur from that time as one "element" in the heap with their summed occurrences. The joining creates a tree growing on while the heap is reducing.

Let's take an example. Given are the characters and occurrences.

a (15) b(7) c(6) d(6) e(5)

In the first step e and d are the rarest characters, so we create this new heap and tree structure:

a(15) de(11) b(7) c(6)

de

/ \

"0"/ \"1"

d e

Next Step:

a(15) bc(13) de(11)

de bc

/ \ / \

"0"/ \"1" "0"/ \"1"

d e b c

Next Step:

a(15) bcde(24)

bcde

/ \

"0"/ \"1"

/ \

de bc

/ \ / \

"0"/ \"1" "0"/ \"1"

d e b c

Next Step unifies the rest:

Huffman-Table

/ \

"0"/ \"1"

/ \

/ \

bcde a

/ \

"0"/ \"1"

/ \

de bc

/ \ / \

"0"/ \"1" "0"/ \"1"

d e b c

Finally this encoding table would be created:

a 1

b 010

c 011

d 000

e 001

Please note, that there is no rule defining what element in the tree is ordered to left or to right [note by buccarel@: in the 'canonical huffman' version, the order does exist and it guarantees that the same suffix tree is generated all the times]. So it's also possible to get e.g. the coding scheme:

a 0

b 100

c 101

d 110

e 111

Taken from CPAN's [Huffman.pm](http://search.cpan.org/~bigj/Algorithm-Huffman-0.02/Huffman.pm)

**Comments**

I tried this question on a recruiting trip to Waterloo and found that almost every student got it pretty easily, which makes this question a little weak when trying to use the results to determine who stays and who goes.

One good thing about the question is that it is a good way to see if the student knows

* how to code a basic algorithm
* what data structure to use as input
* how to traverse a binary tree

[Bstpierr](https://w.amazon.com/bin/view/User:Bstpierr) 23:39, 16 February 2011 (UTC) Waterloo CS students have to implement Huffman encoding as part of their 3rd year data structures class. So that might explain their familiarity with it.

[Branbell](https://w.amazon.com/bin/view/User:Branbell) 01:53, 22 June 2012 (UTC) What is the actual question here? This sounds kind of like "here's an algorithm, write code for it", so it's not really testing much more than the coding competency. Also, having explained this algorithm many times, I know it could easily eat up half the interview just *setting up* the question. Also note: all University of Washington CS students implement this in their second undergrad intro CS course.

**Sample Solution**

Here is a simple implementation in I whipped up in Java.

**import** **java.util.PriorityQueue**;  
   
 **public** **class** **HuffmanExample** {  
   **private** **static** **class** **HuffNode** **implements** Comparable<HuffNode> {  
        **public** HuffNode left  = **null**;  
        **public** HuffNode right = **null**;  
        **public** int      frequency;  
        **public** String   name;  
   
        **public** HuffNode( int frequency, String name ) {  
            **this**.frequency = frequency;  
            **this**.name      = name;  
        }  
   
        **public** HuffNode( HuffNode l, HuffNode r ) {  
            **this**.frequency = l.frequency + r.frequency;  
            **this**.name      = l.name + r.name;  
            **this**.left      = l;  
            **this**.right     = r;  
        }  
   
        **public** int compareTo( HuffNode other ) {  
            **return** Integer.valueOf( **this**.frequency ).compareTo( Integer.valueOf(other.frequency ) );  
        }  
   
        **public** String getEncodings() {  
            **return** HuffNode.getEncodings( **this**, "" );  
        }  
   
        **public** **static** String getEncodings( HuffNode n, String prefix ) {  
            **if**( n.left == **null** && n.right == **null** ) {  
                **return** n.name + "=>" + prefix + "\n";  
            }  
   
            StringBuilder sb = **new** StringBuilder();  
            **if**( n.left != **null** ) {  
                sb.append( getEncodings( n.left, prefix + "0" ) );  
            }  
   
            **if**( n.right != **null** ) {  
                sb.append( getEncodings( n.right, prefix + "1" ) );  
            }  
   
            **return** sb.toString();  
        }  
   
   }  
   
   **private** PriorityQueue<HuffNode> q;  
   
   **public** HuffmanExample(){  
       q = **new** PriorityQueue<HuffNode>();  
   }  
   
   **public** void addNode( HuffNode node ) {  
       q.add( node );  
   }  
   
   **public** String getHuffmanEncoding() {  
       **while**( q.size() > 1 ) {  
           HuffNode a = q.poll();  
           HuffNode b = q.poll();  
           HuffNode r = **new** HuffNode( b, a );  
           q.add( r );  
       }  
   
       **return** q.poll().getEncodings();  
   }  
   
   
   **public** **static** void main( String[] args ) **throws** Exception {  
       HuffmanExample he = **new** HuffmanExample();  
       he.addNode( **new** HuffNode(27, "a" ) );  
       he.addNode( **new** HuffNode(8,  "b" ) );  
       he.addNode( **new** HuffNode(7,  "c" ) );  
       he.addNode( **new** HuffNode(6,  "d" ) );  
       he.addNode( **new** HuffNode(5,  "e" ) );  
   
       System.out.println( he.getHuffmanEncoding() );  
   }  
   
 }

**Question:** Given a pointer to the root of a binary tree write that tree to disk or to a database. Read the tree back in to memory, getting the exact same tree.

You can give the following hints depending on the level:

* This is a Binary Tree not a Binary Search Tree. That means that it is not necessarily sorted; it only means that every node has a degree <= 2.
* Remember you want to get the exact same tree when you read it in.
* You should break this problem into three pieces. a) format in database or file b) write it out c) read it back in

Things to consider:

1. Whatever value you use (int, string, etc), there's nothing in the problem statement that requires the node value to be unique. This can complicate some formats/solutions.
2. Serialization format is not specified. If you get a canned answer, ask them to do it in a different format--one that would not be dependent on order.
3. You can also followup with questions that look for re-usability and design patterns (iterators, visitors, etc).
4. Two trees can have the same pre-order traversal, making serialization/deserialization complex

A

B

C

and

A

B C

Here's a quick-and-dirty solution in Java: [PersistABinaryTree\_JavaSolution1](https://w.amazon.com/bin/view/PersistABinaryTree_JavaSolution1)

I'd definitely say he did a good job then.

How long did it take him?

Scott

> -----Original Message-----

> From: Gabrielson, Jacob

> Sent: Friday, November 21, 2003 4:34 PM

> To: Silver, Scott

> Subject: RE: persist binary tree question

>

>

> Yep, he was able to write the code. The only time he hit a

> snag was when he needed a slight hint to use the associative

> array (I didn't actually say "hash table" or "associative

> array", I just said something along the lines of "store

> somewhere"). It turns out he had actually thought of using a

> hash table, but had discarded it because he thought I'd veto

> it due to memory usage concerns (he should have voiced it

> anyway, but that's another matter). Other than that he got

> the rest of the problem without any hints. So it sounds like

> he actually did relatively well, then? This being the first

> time I've asked the question, I don't have anything to

> calibrate it against.

>

> Thanks,

>

> -- Jacob

>

> -----Original Message-----

> From: Silver, Scott

> Sent: Friday, November 21, 2003 4:28 PM

> To: Gabrielson, Jacob

> Subject: RE: persist binary tree question

>

>

> I think this is a sound solution.

>

> This is basically a variant of the storing a tree in an

> array, A, where for any given node at A[n], we store its left

> child at A[2n] and A[2n+1].

>

> Yep, and you would need a hash (associative array) to read it

> back in. The alternative is to have stored all the NULLs and

> wasted space in the file (and potentially in memory in an

> [non-associative] array).

>

> Was the candidate able to write the code?

>

> Scott

>

>

> > -----Original Message-----

> > From: Gabrielson, Jacob

> > Sent: Friday, November 21, 2003 4:05 PM

> > To: Silver, Scott

> > Subject: RE: persist binary tree question

> >

> >

> > If you have a sec... Have you ever had someone (I just asked

> > this question to someone :-) conceptually assign a number to

> > each "position" in the tree:

> >

> > 1

> > 2 3

> > 4 5 6 7

> >

> > They do a pre-order traversal of the tree, writing out like so:

> >

> > A

> > B

> > C D

> >

> >

> > 1A 3B 6C 7D

> >

> > This makes reading back in a little difficult, but possible

> > (especially if you save a map of numbers -> nodes in, say, a

> > hash table as you're reading back in). How sucky is this

> > solution? It doesn't seem very optimal to me, but I'm

> > wondering if it actually outright breaks in some way I'm not seeing.

> >

> > Thanks,

> >

> > -- Jacob

> >

> > -----Original Message-----

> > From: Silver, Scott

> > Sent: Wednesday, November 19, 2003 10:12 PM

> > To: Gabrielson, Jacob

> > Subject: RE: persist binary tree question

> >

> >

> > I usually say this:

> >

> > "Given a pointer to the root of a binary tree how would write

> > that tree to a file or a database and read that tree back in,

> > getting the exact same tree. I think we should approach this

> > problem in 3 steps. First, let's figure out what the

> > flattened form of the tree looks like in a file or a

> > database. Seconds, let's build a function to write the tree

> > out. Third, let's build a function to read the tree back in"

> >

> > Some people forget about the "exact same tree" part of the

> > question. They often jump to some traversal of a tree.

> > Sometimes when they suggest this, I ask "Can you think of two

> > different trees with the same X-order traversal"? If note,

> > typically the following trees convince them that no single

> > traversal is sufficient. (In fact, you can prove it takes

> > two, but no one has ever done that in an interview).

> >

> > A

> > B

> >

> > And

> >

> > A

> > B

> >

> > Some people try to write just the direction. (ie. A LEFT B).

> > This gets funky if you don't write out the NULL leaves too.

> > Some people try to just write out the parents. Those previous

> > trees should show that isn't sufficient. Some people just try

> > to mark the directions and the non-null leaves. This are two

> > trees that fit this description.

> >

> > Some people forget that the data aren't necessarily unique

> > (so in the database sol'n they don't assign IDs) Some people

> > insist that a binary tree is sorted. (Actually that's a

> > binary search tree) Some people just say they'd use XML. And

> > then it becomes "too easy". You can push them with cyclic graphs.

> >

> > The more interesting part of the question in the reading back in.

> >

> > Scott

> >

> > > -----Original Message-----

> > > From: Gabrielson, Jacob

> > > Sent: Wednesday, November 19, 2003 6:30 PM

> > > To: Silver, Scott

> > > Subject: persist binary tree question

> > >

> > > Hi,

> > >

> > > I'm thinking of asking the persist a binary tree question

> the next

> > > time I do an interview, and was just wondering if there's

> anything

> > > special it would be useful to know about the question.

> Seems pretty

> > > straightforward, but ya' never know.

> > >

> > > Thanks,

> > >

> > > -- Jacob

> > >

> >

>

**Premise**

The problem is to determine the dimensions (x,y) of a grid that is as square as possible and large enough to contain n elements with no more than 2 empty spaces. Orientation of the grid does not matter (tall vs. wide).

Distilled problem statement: given n, minimize |x - y| such that n <= x \* y <= n + 2 and n,y,x are positive integers.

**Highlights**

* General problem solving
* Understanding the problem rules
* Multiple solutions: brute force or optimized
* Loop design and use

**Examples**

* n = 4
  + x = 2, y = 2
  + 0 empty spaces
  + square
* n = 17
  + x = 6, y = 3
  + 1 empty space
  + more square than other options (x = 9, y = 2... etc)
* n = 23
  + x = 5, y = 5
  + 2 empty spaces
  + square, better than other options with fewer empty spaces (x = 6, y = 4)
  + consider withholding this case since the optimal solution has 2 empty spaces
* n = 18
  + x = 5, y = 4
  + 2 empty spaces
  + more square than x = 6, y = 3which has no empty spaces)

**Observations**

The candidate may elect a brute force solution. In doing so it should quickly become apparent to him/her that the solution will consider far too many useless values. For example, when searching for the solution where *n = 17*, the candidate might loop over all possible solutions:

x = 1, y = 17 | x = 2, y = 17 | x = 3, y = 17 | ...

Immediately we see a lot of waste will be considered. So, they might trim *y* while incrementing *x*:

x = 1, y = 17 | x = 2, y = 16 | x = 3, y = 15 | ...

Still, far too many wasteful attempts, and they may not be careful to skip over a potential solution that might be the best one. If these wasteful attempts need to be explained to the candidate that that is a red-flag.

There are many variations on how the candidate will iterate *x* and *y*. Will the candidate find it necessary to use a nested loop? How does the candidate know to exit the loop(s)?

Also, what expression does the candidate propose for determining the "squareness" of the possible solution and whether it's more square than another possible solution?

abs(x1 - y1) < abs(x2 - y2)

If the candidate begins the search at *sqrt(n)* then that's good. But, of course, that won't always be the answer so what does the candidate do to find *y* when *x* = *sqrt(n)*? Discuss performance of the candidate's solution and allow suggestions for improvement.

If you require the candidate implement a function, what does his/her prototype look like for something with 1 input and 2 result values?

Is there an *n* for which the rules cannot be met? - (A1: no...there's always n x 1) *n/2 x 2 provides a better solution for all values of n than n x 1* --[Bartzs](https://w.amazon.com/bin/view/User:Bartzs) 20:37, 15 January 2016 (UTC) - (A2: no...there's always n/3 x 3) *Given that we're always examining a range of 3 consecutive integers, one is bound to be divisible by 3* --[dbronaug](https://w.amazon.com/bin/view/User:dbronaug) 17:30, 21 December 2017 (UTC)

**Solution**

**C**

By [kradek@](https://phonetool.amazon.com/users/kradek)

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

int main(int argc, char\*\* argv) {

if (argc > 1) {

int n = atoi(argv[1]);

int x = (int) sqrt(n);

if (x \* x == n) {

printf("%d x %d = %d\n", x, x, x\*x);

return 0;

}

int y = ++x;

while (x \* y - n > 2) {

y--;

x = (n + y - 1) / y;

}

printf("%d x %d = %d\n", x, y, x\*y);

}

return 0;

}

**awk**

By [ericwith@](https://phonetool.amazon.com/users/ericwith)

function best\_square(n, lo, hi) {

lo = int(sqrt(n));

if (lo \* lo < n) lo++;

hi = lo;

while (lo > 1 && (n > lo \* hi || n + 2 < lo \* hi)) {

if (lo \* hi < n) hi++;

else lo--;

}

print lo "," hi;

}

{ best\_square($1); }

**Java**

By [bartzs@](https://phonetool.amazon.com/users/bartzs)

public static void calculateDimensions(int n) {

int x = (int) Math.sqrt(n);

if (x \* x != n) {

x++;

}

int y = x;

while (x \* y > n + 2) {

x--;

y = (n + x - 1) / x; // (int) Math.ceil((double) n / x)

}

System.out.println(x + "," + y);

}

This may seem like a simple question, and for good candidates it's usually answered fairly easily.

**Premise** - You are given two separate web access log files, say from Apache or some similar web server. The first log file is from day 1, the second one is from day 2. The web site is set up in such a way that every access logged will contain a unique identifier for that customer somewhere in the request line. The website is a fairly high traffic, so these log files are very large.

**Question** - How can you find a unique list of customers who visited on day 1 and then came back for a visit on day 2?

**Follow-up Question** - This question gets much more interesting once you impose memory constraints. What if the set of distinct customers won't fit in memory?

**Solutions**

* Some candidates will start off trying to use grep/sed/awk etc. to pull out the customer ID's... which is good that they're aware of text processing tools and regular expressions, but that's not necessarily the real guts of the problem.
* Other candidates, ignoring the fact that the log files are very large and will still try to read the entire file into memory. In that case, specifically state that the file itself cannot be read into memory, but this is usually a red flag in my mind.
* In some cases, candidates will try to load the data into a database and find duplicates that way. This is not necessarily a bad solution, but if they choose this route, suggest that the number of unique customers is small enough that it can be held in memory, and that no database access should be necessary.
* The "optimal" solution would be to use a hashtable to store the list of unique customers from the first file, and then as you process the second file, check to see if the customer exists in your hashtable. If they do, put their ID into a second hashtable, if not, do nothing. At the end, the second hashtable should contain a list of unique customers that visited on both days. [I think a better solution doesn't use a second hashtable. Why not just use a List to store the \*list\* of repeat visitors? (This requires the candidate to also solve the duplicate problem which is pretty easy if you just modify the first hashtable.) -smithbr]
* The above "optimal" solution has a problem: it doesn't truly scale because it requires everything to fit into memory. Solutions for when you impose memory constraints include sorting each file and then a linear merge comparison. Of course, the candidate needs to describe a sort algorithm that can operate under memory constraints.

**Possible solution**

I actually like a variation of the "database" solution because it doesn't require you to write a script with a hashtable and is achievable just using unix commands. suppose the format of the customer ids, is parseable... then i'd do the following:

sed 's/customer\_id: //g' logFile1 | sort | uniq > day1CustomerIds

sed 's/customer\_id: //g' logFile2 | sort | uniq > day2CustomerIds

join day1CustomerIds day2CustomerIds

Just sort -u would do instead of two process' sort and uniq

sed 's/customer\_id: //g' logFile1 | sort -u > day1CustomerIds

sed 's/customer\_id: //g' logFile2 | sort -u > day2CustomerIds

join day1CustomerIds day2CustomerIds

This has the benefit of actually retaining the list of unique customerIds from day 1 and day 2.

**Scalable Solutions**

If one of the constraints is that the log files are extremely large and cannot fit into memory, one solution is to use a map/reduce approach where the customer IDs are extracted and combined with the date, which is the key then mapped to counts (of 1 initially) and then iteratively reduce the number of unique mappings to join the records and sum up the counts. You could throw this at Hadoop, Spark or come up with a single-host solution.

Another approach might be to extract the ID and date and combine, writing out to a series of files each of which are small enough to be sorted in-memory before being written out, also eliminating uniques as they are found. Those sorted files can be merge-sorted into a single file or an ordered array of files (continue to eliminate uniques), then the final order can be iterated over to determine the number of unique customer IDs. When two customer IDs appear adjacent to each other but with different dates, those are your repeat visitors.

Write a program that reads in a dictionary of words from a file and prompts the user for pairs of words. For each pair, display a shortest possible word chain using the input as the starting and ending words. Print all words in upper case, except the one letter that has changed between words. If either word is not in the dictionary or there is no chain or words, print an appropriate error message.

Two words are connected if there is one letter that different and the rest are the same. For example, MALE connects to MILE (second letter, A to I), but not LIME (first and third letters are different). Here's a sample word ladder for MALL to BENT.

* MALL
* bALL
* BeLL
* BELt
* BEnT

For determining connected words and finding an optimal word chain, describe the runtime for each operation in terms of number of words (N) and letters (L).

(See also [HeadToTailInterviewQuestion](https://w.amazon.com/bin/view/HeadToTailInterviewQuestion).)

Note that, although a good question about CS fundamentals/graph search, this is a very old, well known problem with solutions readily available online in all languages; it even has it's own wikipedia article: <http://en.wikipedia.org/wiki/Word_ladder>

This question should be quick, and it focuses on being detail-oriented and exhaustive in verifying code. It's best done with a whiteboard. A good candidate should complete in 15 minutes, then use this as a warmup for more complicated sorting.

The simple problem is to write a method that sorts three integers.

def threesort(a,b,c):

# illegal, assume no sort method

list = [a,b,c]

list.sort()

return list

def threesort(a,b,c):

# brute force

if (a < b) and (b < c):

return [a,b,c]

if (a < c) and (c < b):

return [a,c,b]

if (b < a) and (a < c):

return [b,a,c]

if (b < c) and (c < a):

return [b,c,a]

if (c < a) and (a < b):

return [c,a,b]

if (c < b) and (b < a):

return [c,b,a]

raise 'programmer error'

def threesort(a,b,c):

# elegant recursive

if not (a < b):

return threesort(b,a,c)

if not (b < c):

return threesort(a,c,b)

return [a,b,c]

def threesort(a,b,c):

# efficient, at most 3 comparisons

if (a < b):

# bubble

if (c > b):

return [a,b,c]

elif (c > a):

return [a,c,b]

else:

return [c,a,b]

else:

# bubble

if (c > a):

return [b,a,c]

elif (c > b):

return [b,c,a]

else:

return [c,b,a]

def assert\_equal(expected, actual):

if expected != actual:

raise 'failed interview'

if \_\_name\_\_ == '\_\_main\_\_':

assert\_equal([1,2,3], threesort(1,2,3))

assert\_equal([1,2,3], threesort(1,3,2))

assert\_equal([1,2,3], threesort(2,1,3))

assert\_equal([1,2,3], threesort(2,3,1))

assert\_equal([1,2,3], threesort(3,1,2))

assert\_equal([1,2,3], threesort(3,2,1))

**Question**

So the Flatland Space Agency is attempting to send two rovers on a mission to Flatland-Mars. The mission requires the two rovers to meet up before they can be effective, but at the last moment the Agency runs out of funding and only has enough money to develop one piece of software that will have to run unchanged on both Rovers.

Each lander lands like the 3d Earth Opportunity and Spirit rovers (lander crashes into the planet with big airbags, rovers emerge from lander, leaving the lander behind) an unknown distance apart.

Problem: Develop a program in the following assembly language which allows the rovers to meet up. Efficiency counts, so try not to require a rover to drive all the way around the planet.

Registers: r0, r1, r2 Instruction set:

* SET <reg> <dec value> (set the register to a decimal number value)
* J0 <reg> <label> (jump to label if the value of the register <reg> is 0)
* JN0 <reg> <label> (jump to label if the value of the register <reg> is not 0)
* BSE <reg> (Tests to see if the rover is in the same place (within 10 units) as a lander base station, if it is put 1 in the specified register, otherwise set it to 0)
* LFT <val> (Move left (counterclockwise) around the planet at specified value units per second for 1 second (assume the execution time for any opcode is 1 second))
* RHT <val> (same as LFT but clockwise)

**Answer**

* The key to this question is understanding that in order to solve it you have to find a condition where the state of the rovers is not the same, so you can break out of the lockstep movement implied by running identical programs
* Best/simplest solution is to go in one direction until hitting a landing site, then increase speed to catch up with the other slower-moving rover. Solution should look something like:

SET r1 0

START:

LFT 10

BSE r0

JN0 r0 CATCHUP

J0 r1 START

CATCHUP:

LFT 100

J0 r1 CATCHUP

**Notes**

* Since you need to get somewhere where the rover's states differ, there is no solution if the rovers land diametrically opposed on the planet. Bonus points if the candidate figures this out.
* The program can also fail if the ratio of the rover's normal and catchup speed is insufficient to catch up before the other lander goes all the way around the planet, finds the other landing site and goes into "catchup" mode as well. Some good math to be explored there, if you've got extra time to fill.

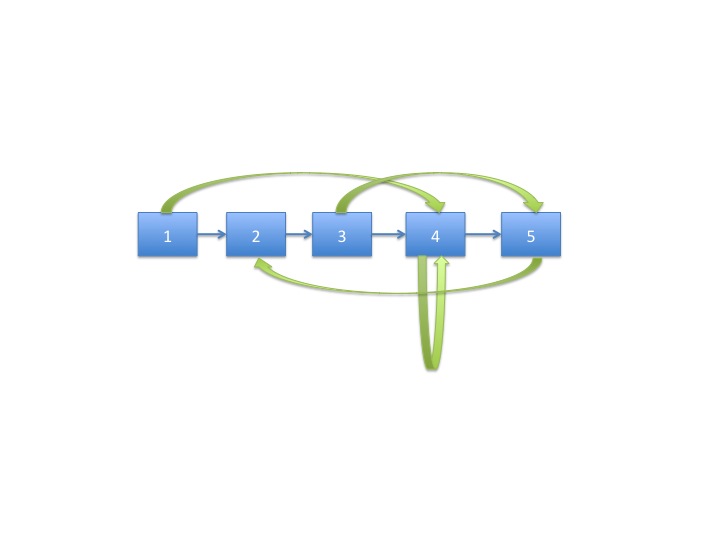
### Problem

Given a linked list where each node has a next pointer and a random pointer, make a deep copy of the list.

### Clarification

The random pointer can be null for some list nodes.

### Example

[](https://improvement-ninjas.amazon.com/s3files/s3get.cgi/randomlinkedlist.jpg)

[From [S3InternalFiles](https://w.amazon.com/bin/view/S3InternalFiles), [Change](https://improvement-ninjas.amazon.com/s3files/s3put.cgi?s3key=randomlinkedlist.jpg)]

### Solution

[joscasti@](https://phonetool.amazon.com/users/joscasti): There's a nice solution in [this page](http://www.geeksforgeeks.org/a-linked-list-with-next-and-arbit-pointer/) (09/2015).

**Background**

Multiplication of a *p*×*q* matrix and another *q*×*r* matrix *pqr* scalar multiplications and *pqr* scalar additions, and results in a *p*×*r* matrix.

Matrix multiplications are also associative, e.g. (*AB*)*C*=*A*(*BC*).

However, given different dimensions of these matrices, different ways of parenthesizing may end up requiring different numbers of scalar operations. Example: Given a 2×300 A, a 300×4 B, and a 4×5 C:

* (*AB*)*C* involves first using 2×300×4=2400 scalar operations to calculate *AB*, a 2×4 matrix, then using 2×4×5=40 more scalar operation for calculating (*AB*)*C*; total: **2440** operations;
* *A*(*BC*) involves 300×4×5=6000 scalar operations for first calculating *BC*, which is a 300×5 matrix, then involves 2×300×5=3000 more scalar operations for calculating *A*(*BC*); total: **9000** operations.

**Question**

Given the dimensions of *N* matrices *Ai*, 1 ≤ *i* ≤ *N*, where the dimensions allow their multiplication, i.e. *A*1*A*2*A*3…*AN*-1*AN* is defined:

**Part 1**

Find all possible ways of parenthesizing them.

**Part 2**

Find the best way to parenthesize them, i.e. so as to minimize the required number of scalar operations for its calculation.

**Give breakdowns of *ABCD* as an example early on**

ABCD can be parenthesized in these ways:

* (((*AB*)*C*)*D*)
* ((*AB*)(*CD*))
* ((*A*(*BC*))*D*)
* (*A*((*BC*)*D*))
* (*A*(*B*(*CD*)))

It helps the candidate in two ways:

* It demonstrates that not every multiplication involves a leaf matrix—((*AB*)(*CD*)) is the counterexample; see the unviable leaf-assimilation approach below.
* The candidate may realize that he had better remember the interim result of parenthesized portions, e.g. each of “(*AB*)”, “(*BC*)”, and “(*CD*)” occurs 3 times above, and this is the first step for a non-brute-force approach.

**Possible approaches the candidate may take**

**Viable approaches**

There are at least four approaches (three practical, one impractical).

**Brute-force, *O*(*N*!)**

It involves a recursive routine whose input is the set of up to N sub-matrices along with the number of operations required so far.

It neither recognizes nor takes advantage of the fact that the same minimum number required for any sub-matrix, e.g. *BCD* out of *ABCDE*, is required no matter whether *A*(*BCD*) or (*BCD*)*E* is the next step.

**Less brute-force but still non-dynamically-programming, *O*(2*N*)**

It involves a recursive routine that takes a sub-sequence of matrices and returns the optimal parenthesized order as well as the number of scalar operations required to compute the result.

It still doesn't take advantage that the output from that function is the same for the same input sub-sequence of matrices (and can be cached, using the input as the key).

**Dynamic-programming, *O*(*N*3)**

This is essentially a caching version of the *O*(2*N*) approach above.  This is also the best one can implement realistically within a 45- or 60-minute loop.  Many people remember this as an example of dynamic programming from their college textbook; our focus should be whether they can come up with a viable implementation (it is fairly straightforward, but not without a few gotchas here and there).

**Polygon triangulation, *O*(*N* log *N*) – IMPRACTICAL FOR INTERVIEWS**

Dated 1981, this is a very advanced approach, which first reduces the problem into a polygon triangulation problem and calculates the optimal way.  Not that many algorithm textbooks introduce this one, but [Wikipedia does](http://en.wikipedia.org/wiki/Matrix_chain_multiplication).  This is also a pretty complex algorithm, and most certainly does not fit the usual interview time budget (less than an hour), so if a candidate says they would implement this, the interviewer should ask whether they can complete it within half an hour.

If they insist upon doing this (+IOHS or maybe -IOHS because this would be “too much”, -BFA, -DR): In my experience, candidates opting for this approach either had a direct knowledge of it, or an (excellently) deep insight that reveals the transformation.  Encouraging the candidate to think out loud is beneficial for seeing which is the case.  One should further question a candidate opting for this approach with the nature of this triangulation—use a pentagon as an example—and see how s/he relates the pentagon to the real answer, so as to make sure his/her understanding is not just textbook-deep.

**Unviable approaches**

**Assimilating leaf matrices**

Multiple candidates tried this one.  They first started with a pair of adjacent nodes as “the core,” e.g. *CD* as in *ABCDEF*, then for each successive iteration, tried to multiply the current core with either the leaf matrix to the left or to the right of the core, e.g. given the current core of *CD*, they compared how many operations *B*(*CD*) and (*CD*)*E* would each take then opt for the one with lesser number of operations as the new core.  This fails to account for possibilities where no such singular core exists, (((*AB*)*C*)(*D*(*EF*))) for example.  Giving 4 matrices—which has one breakdown with two cores, ((*AB*)(*CD*))—or more for the candidate to think about mostly eliminates this pitfall and prevents a false start.

**Problem**

To rotate/shift the elements of a square array clockwise by one at a time. (*Not* by 90 degrees, but by one element. Think of concentric circles.)

Example:

1 2

3 4

becomes:

3 1

4 2

3x3:

1 2 3

4 5 6

7 8 9

becomes:

4 1 2

7 5 3

8 9 6

**Approaches**

* One of the cleanest approach is to temp 0,0, and move the first **column** first, first **row** last. Most candidate do not get time to realize this.
* For starters, candidate should be able to shift the first row successfully.
  + Setting element [0][n] to temp is sufficient. Shift will occur in reverse order; second-last element first, first element last
  + BAD: if they shift first element first, which will create messy overwrite issues, and will need a lot of temporary variables set. Candidate should be able to observe this, and go to the correct solution.
* Once they successfully completed first row and nth column, should not need to complete the rest.
* For inner rotation, either recursion or iteration works fine.
  + Recursion may be more intuitive to most candidates. Either pass in boundaries of smaller array in context of bigger one, OR create a smaller array and pass that to the function.
* Approach should work for 1x1. No need to handle as special case, but fine if they do point out as test case.
* Need to handle 0x0 exception case. Also, need to make sure array is square if creating a function.
* Solid candidate should solve this in 15 minutes max.

**Iterative/C++**

#include <iostream>  
  
**using** **namespace** std;  
  
void  
populateArray( int\*\* numberArray, **const** int & originalSize )  
{  
  int c = 11;  
  
  **for**( int i = 0; i < originalSize; ++i )  
    {  
      **for**( int j = 0; j < originalSize; ++j )  
 {  
  numberArray[i][j] = c++;  
 }  
    }  
}  
  
void  
printArray( int\*\* numberArray, **const** int & originalSize )  
{  
  
  **for**( int i = 0; i < originalSize; ++i )  
    {  
      **for**( int j = 0; j < originalSize; ++j )  
 {  
  cout << numberArray[i][j] << " ";  
 }  
      cout << endl;  
    }  
  
  cout << endl;  
  
}  
  
int  
main(void)  
{  
  
  **while** ( true )  
    {  
  
      int originalSize = 0;  
      cout << "How big? (0 to exit) ";  
      cin >> originalSize;  
  
      **if**( originalSize < 1 )  
 {  
  cout << "Size 0. Aborting." << endl;  
  **return** 0;  
 }  
  
      int attempts = 0;  
      cout << "How many times? ";  
      cin >> attempts;  
  
      // instantiate array  
      int\*\* numberArray = **new** int\* [originalSize];  
  
      **for**( int o = 0; o < originalSize; ++o )  
 {  
  numberArray[o] = **new** int [originalSize];  
 }  
  
      populateArray( numberArray, originalSize );    
      printArray( numberArray, originalSize );  
  
      **const** int startIndexLimit = originalSize / 2;  
  
      **while**( attempts-- > 0 )  
 {  
  
  **for**( int x = 0, endIndexLimit = originalSize;  
       x < startIndexLimit;  
       --endIndexLimit ) // x gets incremented in each loop  
    {  
  
      int y = x;  
  
      // y = x = 0  
  
      int number\_y\_x = numberArray[y][x];  
  
      **while**( y < endIndexLimit - 1 )  
 {  
  numberArray[y][x] = numberArray[++y][x];  
 }  
  
      // y == size - 1  
  
      **while**( x < endIndexLimit - 1)  
 {  
  numberArray[y][x] = numberArray[y][++x];  
 }  
  
      // x == size - 1  
  
      **const** int sizeDifferential = originalSize - endIndexLimit;  
        
      **while**( y > sizeDifferential )  
 {  
  numberArray[y][x] = numberArray[--y][x];  
 }  
        
      // y == 0  
        
      **while**( x > sizeDifferential + 1 ) // up to 1  
 {  
  numberArray[y][x] = numberArray[y][--x];  
 }  
        
      // x == 1  
  
      numberArray[y][x] = number\_y\_x;  
  
    }  
  
  printArray( numberArray, originalSize );  
  
 } // end while; one rotation  
  
      // destroy array  
  
      **for**( int o = 0; o < originalSize; ++o )  
 {  
  **delete** [] numberArray[o];  
 }  
  
      **delete**[] numberArray;  
  
    } // end while; next question  
  
  **return** 0;  
}

**Recursive/Java OOP**

Same basic algorithm as C++ code, written slightly differently.

**class** **RotateArray**  
{  
  
    **private** int originalSize;  
  
    **private** int numberArray[][];  
  
    **public** RotateArray()  
    {  
 **this**( 5 );  
    }  
  
    **public** RotateArray( int originalSize ) **throws** IllegalArgumentException  
    {  
 **if**( originalSize < 1 )  
    {  
 **throw** **new** IllegalArgumentException("Size must be positive.");  
    }  
  
 numberArray = **new** int[originalSize][originalSize];  
 **this**.originalSize = originalSize;  
    }  
  
    **public** void populate()  
    {  
  
 int c = 11;  
   
 **for**( int i = 0; i < originalSize; ++i )  
    {  
 **for**( int j = 0; j < originalSize; ++j )  
    {  
 numberArray[i][j] = c;  
 ++c;  
    }  
    }  
    }  
  
    **public** void rotate()  
    {  
        **this**.rotateByLayer( **this**.originalSize );  
    }  
  
    **private** void rotateByLayer( **final** int currentSize )  
    {  
 **if**( currentSize <= 1 )  
           {  
               **return**;  
           }  
  
 int x = ( **this**.originalSize - currentSize ) / 2;  
        int y = x;  
        **final** int endIndexLimit = x + currentSize;  
  
 // y = x = 0  
  
 int number\_y\_x = numberArray[y][x];  
  
 **for**( ; y < endIndexLimit - 1; ++y )  
    {  
 numberArray[y][x] = numberArray[y+1][x];  
    }  
  
 // y == size - 1  
  
 **for**( ; x < endIndexLimit - 1; ++x )  
    {  
 numberArray[y][x] = numberArray[y][x+1];  
    }  
  
 // x == size - 1  
  
 **final** int sizeDifferential = originalSize - endIndexLimit;  
        
 **for**( ; y > sizeDifferential; --y )  
    {  
 numberArray[y][x] = numberArray[y-1][x];  
    }  
        
 // y == 0  
        
 **for**( ; x > sizeDifferential + 1; --x ) // up to 1  
    {  
 numberArray[y][x] = numberArray[y][x-1];  
    }  
        
 // x == 1  
  
 numberArray[y][x] = number\_y\_x;  
  
        // recurse  
 **this**.rotateByLayer( currentSize - 2 );  
  
     }  
  
  
    **public** String toString()  
    {  
  
 StringBuffer sb = **new** StringBuffer();  
  
 **for**( int i = 0; i < originalSize; ++i )  
    {  
 **for**( int j = 0; j < originalSize; ++j )  
    {  
 sb.append( numberArray[i][j] + " " );  
    }  
 sb.append("\n");  
    }  
 **return** sb.toString();  
    }  
  
    **final** **public** **static** void main(String[] arg)  
    {  
  
 **for**( int testSize = 0; testSize < 7; ++testSize )  
    {  
 **try**  
    {  
 RotateArray numberArray = **new** RotateArray( testSize );  
 numberArray.populate();  
                        System.out.println( "Original array " + testSize + ": ");  
 System.out.println( numberArray );  
 numberArray.rotate();  
                        System.out.println( "Posttransform array " + testSize + ": ");  
 System.out.println( numberArray );  
    }  
 **catch** (IllegalArgumentException iae)  
    {  
 iae.printStackTrace();  
    }  
    }  
    }  
  
}

**With Iterator**

This can be better done using an iterator. I'll have to code that one up sometime.

Here is a solution using an iterator. It does not handle the internal elements. Tested with 2x and 3x arrays as in the example above.

**package** **com.company**;  
  
**import** **java.util.Iterator**;  
  
**public** **class** **Main** {  
  
    **static** **final** int[][] twoXarray = **new** int[][]{  
            {1, 2},  
            {3, 4}};  
  
    **static** **final** int[][] threeXarray = **new** int[][]{  
            {1, 2, 3},  
            {4, 5, 6},  
            {7, 8, 9}};  
  
    **public** **static** void main(String[] args) {  
  
        Main m = **new** Main();  
  
        int [][] rotatedArray = m.rotate(threeXarray, threeXarray.length);  
        printArray(threeXarray, threeXarray.length);  
        System.out.println();  
        printArray(rotatedArray, threeXarray.length);  
    }  
  
    **private** **static** void printArray(int[][] a, int size) {  
        **for** (int i = 0; i < size; ++i) {  
            **for** (int j = 0; j < size; ++j) {  
                System.out.print(a[i][j]);  
                System.out.print(' ');  
            }  
            System.out.print('\n');  
        }  
    }  
  
    **private** int[][] rotate(int[][] array, int size) {  
  
        int[][] target = **new** int[size][size];  
  
        Clockwise2DArrayPositionIterator sourceItr = **new** Clockwise2DArrayPositionIterator(size);  
        Clockwise2DArrayPositionIterator targetItr = **new** Clockwise2DArrayPositionIterator(size);  
  
        Coord firstVal = targetItr.next();  
  
        **while** (targetItr.hasNext()) {  
            Coord sourceCoord = sourceItr.next();  
            Coord targetCoord = targetItr.next();  
  
            target[targetCoord.y][targetCoord.x] = array[sourceCoord.y][sourceCoord.x];  
        }  
  
        Coord lastVal = sourceItr.next();  
        target[firstVal.y][firstVal.x] = array[lastVal.y][lastVal.x];  
  
        **return** target;  
    }  
  
    **static** **class** **Coord** {  
        **final** int x, y;  
  
        Coord(int x, int y) {  
            **this**.x = x;  
            **this**.y = y;  
        }  
  
        @Override  
        **public** String toString() {  
            **return** "[" + x + "," + y + "]";  
        }  
    }  
  
    /\*\*  
     \* Given a square array, list off coords in clockwise order.  
     \*/  
    **static** **class** **Clockwise2DArrayPositionIterator** **implements** Iterator<Coord> {  
  
        **private** int size;  
  
        **enum** Segment {  
            NORTH, EAST, SOUTH, WEST;  
        }  
  
        int x = 0, y = 0;  
        boolean started = **false**;  
        Segment currentSegment = Segment.NORTH;  
  
        **public** Clockwise2DArrayPositionIterator(int size) {  
            **this**.size = size;  
        }  
  
        @Override  
        **public** boolean hasNext() {  
            **if** (x == 0 && y == 0 && started) {  
                **return** **false**;  
            }  
  
            **return** **true**;  
        }  
  
        @Override  
        **public** Coord next() {  
            started = **true**;  
  
            Coord nextCoord = **new** Coord(x, y);  
  
            **if** (x == size - 1 && y == 0 ) {  
                currentSegment = Segment.EAST;  
            } **else** **if** (x == size -1 && y == size - 1) {  
                currentSegment = Segment.SOUTH;  
            } **else** **if** (x == 0 && y == size - 1) {  
                currentSegment = Segment.WEST;  
            }  
  
            **switch** (currentSegment) {  
                **case** NORTH:  
                    x++;  
                    **break**;  
                **case** EAST:  
                    y++;  
                    **break**;  
                **case** SOUTH:  
                    x--;  
                    **break**;  
                **case** WEST:  
                    y--;  
                    **break**;  
            }  
  
            **return** nextCoord;  
        }  
  
        @Override  
        **public** void remove() {  
            **throw** **new** IllegalArgumentException();  
        }  
    }  
}

### Purpose

After searching for just the right problem complexity for SDE interview questions, I personally developed this coding problem for usage during SDE Interviews. It purposely does not use higher-level data structures which might ordinarily be used, like HashSet to guarantee uniqueness of values. In fact, the code solution shown here leverages nothing but Java Arrays. The focus of the problem is purely logical, with the goal of determining how the interviewee thinks about problems.

To date (Aug 2019),  I have used this as an interview question over 20 times with good results. The code has several optional layers which can be added in case the candidate solves the problem quickly. For instance, these methods are optional and are normally not required for a standard 25~30 minute coding session:

        sudoku.printGrid(sudokuGrid);  // prints the input grid  
        sudoku.validateChars(sudokuGrid);  // validates that the int grid contains only the allowed chars (digits 1 through 9)

When running a phone screen, I normally cut/paste the top comments section that contains the game rules (i.e. constraints) into the livecode window. Alternately, for in-person interviews, you can write these rules on the board or just state them, according to candidate's familiarity with the game.

As long as the code solves the primary challenge, to validate whether the rules of the Sudoku Game have been met by the input grid, then the interview should be counted a success.

Several times, candidates have pushed back on the need to validate the grid using all of the stated constraints. They have claimed there are shortcuts to arriving at the answer. Therefore, I have found the following discussion on Google Groups that talks about some potential shortcuts for solving Sudoku puzzles:

<https://groups.google.com/forum/#!topic/rec.puzzles/6AFT8aPHZ1E>

There is a proof that shows you can sufficiently test the validity by evaluating only 21 of the 27 stated constraints. However, the solution is actually harder to implement in code than simply walking through all the stated rules. Another claim is that if you simply add all the numbers in each row and column and arrive at a sum of 45 (= 1+2+3+4+5+6+7+8+9), then it is sufficient proof. However, I have proven via coding test that this alone does not provide sufficiency.

### Sample Code

**package** **com.amazon**;  
  
**import** **java.util.Arrays**;  
  
/\*\*  
 \* Created by phhargis on 5/22/18.  
 \*  
 \* Interview Problem:  Sudoku Validator  
 \*  
 \* Requirements:  
 \*   each row contains digits 1 to 9, each digit occurs only once  
 \*   each col contains digits 1 to 9, each digit occurs only once  
 \*   each sub-grid (3x3) contains digits 1 to 9, each digit occurs only once  
 \*  
 \* For example, here is a valid grid which should return true:  
 \*  
 \* int[][] sudokuGrid = {  
 \*        {5,3,4,6,7,8,9,1,2},  
 \*        {6,7,2,1,9,5,3,4,8},  
 \*        {1,9,8,3,4,2,5,6,7},  
 \*        {8,5,9,7,6,1,4,2,3},  
 \*        {4,2,6,8,5,3,7,9,1},  
 \*        {7,1,3,9,2,4,8,5,6},  
 \*        {9,6,1,5,3,7,2,8,4},  
 \*        {2,8,7,4,1,9,6,3,5},  
 \*        {3,4,5,2,8,6,1,7,9}  
 \*\*/  
**public** **class** **Sudoku** {  
  
   /\*\*  
     \* Let's generate a Sudoku validator  
     \*\*/  
   **private** **static** **final** int GRIDSIZE = 9;  
  
   **public** **static** boolean validate(int[][] grid) {  
  
       **for** (int i = 0; i < GRIDSIZE; i++) {  
           **if** (!validateRow(grid, i)) {  
               **return** **false**;  
           }  
  
           **if** (!validateCol(grid, i)) {  
               **return** **false**;  
           }  
       }  
  
       **for** (int i = 0; i < GRIDSIZE; i += 3) {  
           **for** (int j = 0; j < GRIDSIZE; j += 3) {  
               **if** (!validateSubGrid(grid, i, j)) {  
                   **return** **false**;  
               }  
           }  
       }  
  
       **return** **true**;  
   }  
  
   **public** **static** boolean validateRow(int[][] grid, int rowIdx) {  
       int[] counts = **new** int[GRIDSIZE];  
       **for** (int colIdx = 0; colIdx < GRIDSIZE; colIdx++) {  
            counts[grid[rowIdx][colIdx] - 1]++;  
       }  
  
       **return** validateCounts(counts);  
   }  
  
   **public** **static** boolean validateCol(int[][] grid, int colIdx) {  
       int[] counts = **new** int[GRIDSIZE];  
       **for** (int rowIdx = 0; rowIdx < GRIDSIZE; rowIdx++) {  
            counts[grid[rowIdx][colIdx] - 1]++;  
       }  
  
       **return** validateCounts(counts);  
   }  
  
   **public** **static** boolean validateSubGrid(int[][] grid, int rowOffset, int colOffset) {  
       int[] counts = **new** int[GRIDSIZE];  
  
       **for** (int i = rowOffset; i < rowOffset + 3; i++) {  
           **for** (int j = colOffset; j < colOffset + 3; j++) {  
                counts[grid[i][j] - 1]++;  
           }  
       }  
  
       **return** validateCounts(counts);  
   }  
  
   **public** **static** void validateChars(int[][] intGrid) {  
       **for** (int i = 0; i < GRIDSIZE; i++) {  
           **for** (int j = 0; j < GRIDSIZE; j++) {  
               char c = (String.valueOf(intGrid[i][j])).charAt(0);  
               **if** (c < '1' || c > '9') {  
                   **throw** **new** RuntimeException("Bad character");  
               }  
           }  
       }  
   }  
  
   **public** **static** boolean validateCounts(int[] counts) {  
       **return** Arrays.stream(counts).filter(count -> count == 1).count() == GRIDSIZE;  
   }  
  
   **public** void printGrid(int[][] grid) {  
       **final** String comma = ",";  
        System.out.println("Sudoku Grid:");  
       **for**(int row=0; row<GRIDSIZE; row++) {  
            System.out.print("{");  
           **for** (int col = 0; col < GRIDSIZE; col++) {  
                String msg = grid[row][col] + (col<8 ? comma:"");  
                System.out.print(msg);  
           }  
            System.out.println("}");  
       }  
   }  
  
   **public** **static** void main(String[] args) {  
       int[][] sudokuGrid = {  
           {5, 3, 4, 6, 7, 8, 9, 1, 2},  
           {6, 7, 2, 1, 9, 5, 3, 4, 8},  
           {1, 9, 8, 3, 4, 2, 5, 6, 7},  
           {8, 5, 9, 7, 6, 1, 4, 2, 3},  
           {4, 2, 6, 8, 5, 3, 7, 9, 1},  
           {7, 1, 3, 9, 2, 4, 8, 5, 6},  
           {9, 6, 1, 5, 3, 7, 2, 8, 4},  
           {2, 8, 7, 4, 1, 9, 6, 3, 5},  
           {3, 4, 5, 2, 8, 6, 1, 7, 9}  
       };  
  
        Sudoku sudoku = **new** Sudoku();  
        sudoku.printGrid(sudokuGrid);  
        sudoku.validateChars(sudokuGrid);  
       boolean answer = sudoku.validate(sudokuGrid);  
        System.out.println("Is Sudoku Grid valid: " + answer);  
   }  
  
}