19 Essential Algorithm Interview Questions *

Looking for experts? Check out Toptal's algorithm developers.

What are Divide and Conquer algorithms? Describe how they work. Can you give any common examples of the types of problems where this approach might be used?

View the answer →Hide answer

Divide and Conquer algorithms are a paradigm for solving problems that involve several basic steps. First, we divide the problem into smaller pieces and work to solve each of them independently. Once we've solved all of the pieces, we take all of the resulting smaller solutions and combine them into a single integrated comprehensive solution.

This process can be performed recursively; that is, each "sub problem" can itself be subdivided into even smaller parts if necessary.. This recursive division of the is performed until each individual problem is small enough to become relatively trivial to solve.

Some common examples of problems that lend themselves well to this approach are binary search, sorting algorithms (e.g., Merge Sort, Quicksort), optimization of computationally complex mathematical operations (Exponentiation, FFT, Strassen's algorithm), and others.

How would you optimally calculate p^k, where k is a non-negative integer? What is the complexity of the solution?

View the answer →Hide answer

First, let's mention that the trivial solution has the complexity of O(k). The problem can be solved by squaring and multiplying.

We know that p^k = p^x * p^y if x+y=k. We also know that p^k = (p^a)^b if a*b=k.

For an even value of k, choosing a = 2 and b = k/2, thus having p^k = (p^2)^(k/2), will reduce the number of required multiplications almost in half. For an odd value of k, choosing x = 1 and y=k-1 will result in y being even, and we can then simply repeat the same process as for the even case. This allows us to define a recursive function:

```
FUNCTION pow(base, exponent)
        IF exponent == 0
                RETURN 1
        ELSE IF exponent is even
                RETURN pow(base * base, exponent / 2)
        ELSE
                RETURN base * pow(base * base, (exponent - 1) / 2)
        END IF
```

This solution results in a complexity of O(log k).

How do Insertion sort, Heapsort, Quicksort, and Merge sort work?

View the answer →Hide answer

Insertion sort takes elements of the array sequentially, and maintains a sorted subarray to the left of the current point. It does this by taking an element, finding its correct position in the sorted array, and shifting all following elements by 1, leaving a space for the element to be inserted.

Heapsort starts by building a max heap. A binary max heap is a nearly complete binary tree in which each parent node is larger or equal to its children. The heap is stored in the same memory in which the original array elements are. Once the heap is formed, it completely replaces the array. After that, we take and remove the first element, restore the heap property, thus reducing the heap size by 1, after which we place the max element at the end of that memory. This is repeated until we empty out the heap, resulting in the smallest element being in the first place, and the following elements being sequentially larger.

Quicksort is performed by taking the first (leftmost) element of the array as a pivot point. We then compare it to each following element. When we find one that is smaller, we move it to the left. The moving is performed quickly by swapping that element with the first element after the pivot point, and then swapping the pivot point with the element after it. After going through the whole array, we take all points on the left of the pivot and call quicksort on that subarray, and we do the same to all points on the right of the pivot. The recursion is performed until we reach subarrays of 0-1 elements in length.

Merge sort recursively halves the given array. Once the subarrays reach trivial length, merging begins. Merging takes the smallest element between two adjacent subarrays and repeats that step until all elements are taken, resulting in a sorted subarray. The process is repeated on pairs of adjacent subarrays until we arrive at the starting array, but sorted.

Check out the visualization of these sorting algorithms here: www.sorting-algorithms.com

What are the key advantages of Insertion Sort, Quicksort, Heapsort and Mergesort? Discuss best, average, and worst case time and memory complexity.

View the answer →Hide answer

Insertion sort has an average and worst runtime of O(n^2), and a best runtime of O(n). It doesn't need any extra buffer, so space complexity is O(1). It is efficient at sorting extremely short arrays due to a very low constant factor in its complexity. It is also extremely good at sorting arrays that are already "almost" sorted. A common use is for re-sorting arrays that have had some small updates to their elements.

The other three algorithms have a best and average runtime complexity of O(n log n). **Heapsort** and **Mergesort** maintain that complexity even in worst case scenarios, while Quicksort has worst case performance of O(n^2).

Quicksort is sensitive to the data provided. Without usage of random pivots, it uses O(n^2) time for sorting a full sorted array. But by swapping random unsorted elements with the first element, and sorting afterwards, the algorithm becomes less sensitive to data would otherwise cause worst-case behavior (e.g. already sorted arrays). Even though it doesn't have lower complexity than **Heapsort** or **Merge sort**, it has a very low constant factor to its execution speed, which generally gives it a speed advantage when working with lots of random data.

Heapsort has reliable time complexity and doesn't require any extra buffer space. As a result, it is useful in software that requires reliable speed over optimal average runtime, and/or has limited memory to operate with the data. Thus, systems with real time requirements and memory constraints benefit the most from this algorithm.

Merge sort has a much smaller constant factor than Heapsort, but requires O(n) buffer space to store intermediate data, which is very expensive. Its main selling point is that it is stable, as compared to Heapsort which isn't. In addition, its implementation is very parallelizable.

What is a Hash Table, and what is the average case and worst case time for each of its operations? How can we use this structure to find all anagrams in a dictionary?

View the answer →Hide answer

A Hash Table is a data structure for mapping values to keys of arbitrary type. The Hash Table consists of a sequentially enumerated array of buckets of a certain length. We assign each possible key to a bucket by using a hash function - a function that returns an integer number (the index of a bucket) for any given key. Multiple keys can be assigned to the same bucket, so all the (key, value) pairs are stored in lists within their respective buckets.

Choosing the right hashing function can have a great impact on performance. A hash function that is good for a dataset that we want to store will result in hashes of different keys being a rare occurrence. Even though accessing, inserting and deleting have a worst case time complexity of O(N) (where N is the number of elements in the Hash Table), in practice we have an average time complexity of O(1).

To find all anagrams in a dictionary, we just have to group all words that have the same set of letters in them. So, if we map words to strings representing their sorted letters, we could group words into lists by using their sorted letters as a key.

```
FUNCTION find_anagrams(words)
        word_groups = HashTable<String, List>
        FOR word IN words
                word_groups.get_or_default(sort(word), []).push(word)
        END FOR
        anagrams = List
        FOR key, value IN word_groups
                anagrams.push(value)
        END FOR
        RETURN anagrams
```

The hash table stores lists mapped to strings. For each word, we add it to the list at the appropriate key, or make a new list and add it to it then. On average, for a dictionary of N words of length less or equal to L, this algorithm works with an average time complexity of O(N L log L).

A numeric array of length N is given. We need to design a function that finds all positive numbers in the array that have their opposites in it as well. Describe approaches for solving optimal worst case and optimal average case performance, respectively.

View the answer →Hide answer

Let us first design an approach with optimal worst case time.

We need to compare numbers to see if they have their opposites in the array. The trivial solution of comparing all numbers has a consistent time of O(N^2). The great number of comparisons involved should suggest trying to establish a total order operator that allows us to use sorting for solving the problem. If we define a comparison operator that places all instances of a number right after all instances of its opposite, we would have exactly pair of consecutive opposite numbers for each number that has its opposite in the array.

An example of what we want to achieve:

```
Array: -7 4 -3 2 2 -8 -2 3 3 7 -2 3 -2
Sorted: -2 -2 -2 2 2 -3 3 3 4 -7 7 -8
```

We see that after our special sorting method, we have [-2, 2], [-3, 3] and [-7, 7] combinations happening consecutively exactly once. Implementing this comparison is simple and it can be implemented as follows.

```
FUNCTION compare(a, b)
        IF a != b and a != -b
                RETURN abs(a) < abs(b)
        ELSE
                RETURN a < b
```

If the numbers aren't equal or opposite, we sort them by their absolute value, but if they are, we sort them by their sign. Finally, a solution based on this is now very simple:

```
FUNCTION find_numbers_with_opposites(numbers)
        answer = List
        sorted_numbers = sort_by(numbers, compare)
        FOR n IN [1..sorted_numbers.length()]
                IF sorted_numbers[n] > 0 AND sorted_numbers[n - 1] == -sorted_numbers[n]
                        answer.push(n)
                END IF
        END FOR
        RETURN answer
```

This implementation has a worst case runtime complexity of O(N log N), with the sorting algorithm being the bottleneck.

Optimal average case time complexity of O(N) can be achieved using Hash Tables. We map numbers to their absolute values, and check if their opposites are already in the Hash Table.

```
FUNCTION find_numbers_with_opposites(numbers)
        table = HashTable<number, number>
        answer = List
        FOR number IN numbers
                IF number == 0
                        CONTINUE
                END IF
                key = abs(number)
                IF key not in table
                        table[key] = number
                ELSE IF table[key] = -number
                        answer.push(key)
                        table[key] = 0
                END IF
        END FOR
```

We change the value in the table to something that will never be equal to any of the numbers in the array so we don't return duplicate results from duplicate matches.

All HashTable operations have an average time complexity of O(1), and our complexity is a result of performing operations N times.

How would you, in general terms, describe dynamic programming? As an example, how would you find the length of the longest common subsequence of elements in two arrays by using this method?

View the answer →Hide answer

Dynamic programming is a paradigm for solving optimization problems. It consists of finding solutions for intermediate subproblems, which can be stored and reused for solving the actual problem. Dynamic programming is the best approach for difficult problems that always become trivial once we know the solution for a slightly easier instance of that problem - the intermediate subproblem. Dynamic programming solutions are best represented by a recursive relation, and easily implemented.

If the intermediate subproblems are not overlapping, then we have just a case of Divide and Conquer.

Finding the longest common subsequence (LCS) between two arrays is a classical example of using dynamic programming. Let the arrays have lengths M and N, and stored as variables a[0:M] and b[0:N]. Let's use L[p, q] to mark the length of the LCS for subarrays a[0:p] and b[0:q]; that is, L[p, q] == LCS(a[0:p], b[0:q]). Let's also visualize what a matrix L[p, q] would look like for an example pair of "bananas" and "sandal".

p/q	0	1 B	2 A	3 N	4 A	5 N	6 A	7 S
0	0	0	0	0	0	0	0	0
1 S	0	0	0	0	0	0	0	1
2 A	0	0	1	1	1	1	1	1
3 N	0	0	1	2	2	2	2	2
4 D	0	0	1	2	2	2	2	2
5 A	0	0	1	2	3	3	3	3
6 L	0	0	1	2	3	3	3	3

If p or q is zero, then L[p, q] = 0 since we have one empty subarray. All other fields have a simple rule connecting them - L[p, q] equals to the maximum value of the following options:

- L[p - 1, q] - the LCS didn't change, we just added one letter to array a to achieve L[p, q]
- L[p, q - 1] - analogous for array b
- L[p - 1, q - 1]+1 - adding the same letter to both a and b, which of course can't happen for every field

If you look at the table again, you can see that numbers are always equal to the maximum of their upper or left neighbor, unless the values in that field are equal, in which case they increment that maximum by 1. So a solution to the problem is given with the following algorithm.

```
FUNCTION lcs(a, b)
        M = a.length()
        N = b.length()
        L = Matrix[M + 1, N + 1]
        FOR i IN [0..M]
                L[i, 0] = 0
        END FOR
        FOR i IN [0..N]
                L[0, i] = 0
        END FOR
        FOR i IN [1..M]
                FOR j IN [1..N]
                        L[i, j] = max(L[i-1, j], L[i, j-1])
                        IF a[i-1] == b[j-1]
                                L[i, j] = max(L[i, j], L[i-1, j-1] + 1)
                        END IF
                END FOR
        END FOR
        RETURN L[M, N]
```

The time complexity of this solution is O(MxN)

Design an algorithm that finds the number of ways in which you can traverse N meters by doing jumps of 1, 2, 3, 4, or 5 meter lengths. Assume that N can be a very large number. What is the resulting complexity?

View the answer →Hide answer

We can use dynamic programming for solving this problem. Let's use n[k] to represent the number of ways we can reach distance k. That distance can be reached by jumping from one of the 5 previous distances. Thus the number of ways in which we can reach this distance is the sum of the ways in which we can reach the previous 5 distances:

n[k] = n[k-1] + n[k-2] + n[k-3] + n[k-4] + n[k-5]

The solution is a simple for loop.

```
FUNCTION ways(N)
        Array n[N+1]
        n[0] = 1
        FOR k IN [1..N]
                n[k] = 0
                FOR d IN [1..min(5, k)+1]
                        n[k] += n[k - d]
                END FOR
        END FOR
        RETURN n[N]
```

This solution has a time complexity of O(N). But, we can have even better performance. The given sum can be represented as a 1x5 matrix of ones multiplied by a 5x1 matrix of previous elements. If we use the same approach for shifting, we can get the relation B[k] = A * B[k-1], where:

```
B[k] =
        [n[k-4] ]
        [n[k-3] ]
        [n[k-2] ]
        [n[k-1] ]
        [n[k]   ]
A =
        [0      1       0       0       0]
        [0      0       1       0       0]
        [0      0       0       1       0]
        [0      0       0       0       1]
        [1      1       1       1       1]
```

If B[0] = [0 0 0 0 1]', then [0 0 0 0 1] * B[k] = n[k]. Now, due to B[k] = A * B[k-1], B[k] = A^T * B[0]. With that, the solution of our problem can be represented as a relation n[N] = [0 0 0 0 1] * A^N * [0 0 0 0 1]'. If we use the previously mentioned optimal approach for calculating pow(A, N), this solution has an O(log N) time complexity. We have to keep in mind that this does have a high constant bound to this complexity, since matrix multiplication takes time. But, for a large enough N, this solution is optimal.

What are Red-Black Trees and B-Trees? What is the best use case for each of them?

View the answer →Hide answer

Both Red-Black Trees and B-Trees are balanced search trees that can be used on items that can have a comparison operator defined on them. They allow operations like minimum, maximum, predecessor, successor, insert, and delete, in O(log N) time (with N being the number of elements). Thus, they can be used for implementing a map, priority queue, or index of a database, to name a few examples.

Red-Black Trees are an improvement upon Binary Search Trees. Binary Search Trees use binary trees to perform the named operations, but the depth of the tree is not controlled - so operations can end up taking a lot more time than expected. Red-Black Trees solve this issue by marking all nodes in the tree as red or black, and setting rules of how certain positions between nodes should be processed. Without going into too much detail, this method guarantees that the longest branch isn't more than twice as long as the shortest branch, so each branch is shorter than 2*log_base2(N).

This is the ideal structure for implementing ordered maps and priority queues. B-Trees branch into K-2K branches (for a given number K) rather than into 2, as is typical for binary trees. Other than that, they behave pretty similarly to a binary search tree. This has the advantage of reducing access operations, which is particularly useful when data is stored on secondary storage or in a remote location. This way, we can request data in larger chunks, and by the time we finish processing a previous request, our new request is ready to be handled. This structure is often used in implementing databases, since they have a lot of secondary storage access.

What are the Dijkstra and Prim algorithms, and how are they implemented? How does the Fibonacci heap relate to them?

View the answer →Hide answer

Dijkstra is an algorithm for finding single source shortest paths. Prim is an algorithm for finding minimum spanning trees.

Both algorithms have a similar implementation. We'll demonstrate just calculating the distance and spanning tree size, but the shape of the solutions can be easily derived.

```
FUNCTION dijkstra(graph, start_point, end_point)
        heap = MinHeap
        visited = Array[graph.node_count()]
        heap.add(Connection(start_point, 0))
        WHILE !heap.empty()
                n = heap.pop()
                IF visited[n.point]
                        CONTINUE
                END IF
                IF n.point == end_point
                        RETURN n.distance
                END IF
                visited[n.point] = true
                FOR child IN graph[n.point].children()
                        IF !visited[child.point]
                                heap.add(Connection(child.point, n.distance + child.distance)
                        END IF
                END FOR
        END WHILE
        RETURN ERROR("No path found")

FUNCTION prim(graph)
        heap = MinHeap
        tree_size = 0
        tree_count = 0
        visited = Array[graph.node_count()]
        heap.add(Connection(0, 0))
        WHILE !heap.empty()
                n = heap.pop()
                IF visited[n.point]
                        CONTINUE
                END IF
                tree_size += n.distance
                tree_count += 1
                visited[n.point] = true
                FOR child IN graph[n.point].children()
                        IF !visited[child.point]
                                heap.add(Connection(child.point, child.distance)
                        END IF
                END FOR
        END WHILE
        IF tree_count != graph.node_count()
                RETURN ERROR("Graph is not connected")
        ELSE
                RETURN tree_size
        END IF
```

If you look closely, you'll see that the only difference (besides the return data calculations) is the data added to the heap: while Dijkstra accumulates the distance, Prim just uses the distance of the branch. Both algorithms form a tree by taking branches with the smallest price from the MinHeap. In Dijkstra, the point closest to the starting point has the smallest price, while in Prim the point closest to their parent has the smallest price.

Using a normal binary heap, we can't prevent adding duplicates. Thus, the heap can have as many item additions as there are edges in the graph. If V represents the number of vertices, while E the number of edges, then the complexity is O((E+V) log V).

The bottleneck is the fact that, in the worst case, we will add all edges to the heap at some point. Multiple edges can point to one vertex, so all but one edge pointing to that vertex will be thrown away in the visited check. To prevent that, we can allow the heap to directly access the element, update the edge if it's better, and then heapify to fix any changes to the order. This operation has the complexity of O(log V). In a Fibonacci Heap this operation has an O(1) complexity. Thus by using a Fibonacci heap this complexity can be reduced to O(E + V log V).

What is the Bellman-Ford algorithm for finding single source shortest paths? What are its main advantages over Dijkstra?

View the answer →Hide answer

The Bellman-Ford algorithm finds single source shortest paths by repeatedly relaxing distances until there are no more distances to relax. Relaxing distances is done by checking if an intermediate point provides a better path than the currently chosen path. After a number of iterations that is slightly less than the node count, we can check if the solution is optimal. If not, there is a cycle of negative edges that will provide better paths infinitely long.

This algorithm has the advantage over Dijkstra because it can handle graphs with negative edges, while Dijkstra is limited to non-negative ones. The only limitation it has are graphs with cycles that have an overall negative path, but this would just mean that there is no finite solution.

Let's again just implement finding a distance, rather than the actual path that it represents:

```
FUNCTION bellman_ford(graph, start_node, end_node)
        dist = Array[graph.node_count()]
        FOR n IN [0..graph.node_count()]
                dist[n] = infinity
        END FOR
        updated = False
        FOR x in [0..graph.node_count()]
                updated = false
                FOR n in [0..graph.node_count()]
                        FOR p IN graph[n].parents()
                                new_distance = dist[p.point] + p.distance
                                IF dist[n] > new_distance
                                        dist[n] = new_distance
                                        updated = true
                                END IF
                        END FOR
                END FOR
                IF !updated
                        BREAK
                END IF
        END FOR
        IF updated
                RETURN ERROR("Contains negative cycles, unsolvable")
        ELSE IF dist[end_node] == infinity
                RETURN ERROR("There is no connection between the start and end node")
        ELSE
                RETURN dist[end_node]
        END IF
```

The complexity of this algorithm is O(V * E), which is slower than Dijkstra in most cases. So this algorithm should be used only when we expect negative edges to exist.

What is A*, what are its implementation details, and what are its advantages and drawbacks in regard to traversing graphs towards a target?

View the answer →Hide answer

A* is an algorithm for pathfinding that doesn't attempt to find optimal solutions, but only tries to find solutions quickly and without wandering too much into unimportant parts of the graph.

It does this by employing a heuristic that approximates the distance of a node from the goal node. This is most trivially explained on a graph that represents a path mesh in space. If our goal is to find a path from point A to point B, we could set the heuristic to be the Euclidean distance from the queried point to point B, scaled by a chosen factor.

This heuristic is employed by adding it to our distance from the start point. Beyond that, the rest of the implementation is identical to Dijkstra.

The algorithm's main advantage is the quick average runtime. The main disadvantage is the fact that it doesn't find optimal solutions, but any solution that fits the heuristic.

We are given an array of numbers. How would we find the sum of a certain subarray? How could we query an arbitrary number of times for the sum of any subarray? If we wanted to be able to update the array in between sum queries, what would be the optimal solution then? What's the preprocessing and query complexity for each solution.

View the answer →Hide answer

For the sake of notation, let us represent the length of the array as N.

The first problem consists of calculating the sum of the array. There is no preprocessing involved and we do one summing operation of O(N) complexity.

The second problem needs to calculates sums multiple times. Thus, it would be wise to perform preprocessing to reduce the complexity of each query. Let's replace the create an array of subsums s[0:N+1] for the array a[0:N], that is:

```
s[0] = 0
FOR k in [1..N+1]
        s[k] = s[k-1] + a[k-1]
END FOR
```

Now each element k stores the sum of a[0:k]. To query the sum of a subarray a[p:q], to take the sum of all elements until q s[q] and subtract the sum of all elements before p s[p], that is subsum(p, q) = s[q] - s[p]

The preprocessing for this method takes O(N) time, but each query takes O(1) time to perform.

The hardest problem is responding to an arbitrary number of data updates and queries. First, let us look at the previous solutions. The first solution has O(1) insertion complexity, but O(N) query complexity. The second solution has the opposite, O(N) insertion and O(1) queries. Neither of these approaches is ideal for the general case. Ideally, we want to achieve a low complexity for both operations.

A **Fenwick tree (or binary indexed tree)** is ideal for this problem. We maintain an array tree[0:N+1] of values, where every N-th item stores the sum(a[M:N]), and where M is equal to N with the least significant 1 in its binary representation replaced by 0. So for example, N = 19 = b10011, M = 18=b10010; N = 20 = b10100, M=16 = b10000. Now we can easily calculate the sum by following M until we reach 0. Updates are done in the opposite direction.

```
A = Array[N]
Tree = Array[N+1]

FUNCTION sum(end)
        result = 0
        WHILE end > 0
                result += tree[end]
                last_one = end & -end
                end -= last_one
        END WHILE
        RETURN result

FUNCTION update(index, value)
        increment = value - a[index]
        WHILE index < tree.length()
                tree[index] += increment
                last_one = index & -index
                index += last_one
        END WHILE

FUNCTION query(p, q)
        RETURN sum(q) - sum(p)
```

Both operations require O(log N) complexity, which is better than either previous approach.

You need to design a scheduler that to schedule a set of tasks. A number of the tasks need to wait for some other tasks to complete prior to running themselves. What algorithm could we use to design the schedule and how would we implement it?

View the answer →Hide answer

What we need to do is a topological sort. We connect a graph of all the task dependencies. We then mark the number of dependencies for each node and add nodes with zero dependencies to a queue. As we take nodes from that queue, we remove a dependency from all of its children. As nodes reach zero dependencies, we add them to the queue.

After the algorithm executes, if the list doesn't have as many elements as the number of tasks, we have circular dependencies. Otherwise, we have a solution:

```
FUNCTION schedule(nodes)
        dependencies = Array[nodes.length()]
        FOR node IN nodes
                FOR child IN node.children
                        dependencies[child] += 1
                END FOR
        END FOR
        queue = Queue
        solution = List
        FOR n IN [0..nodes.length()]
                IF dependencies[n] == 0
                        queue.push(n)
                END FOR
        END FOR
        WHILE !queue.empty()
                node = queue.pop()
                solution.push(node)
                FOR n IN nodes[node].children
                        dependencies[n] -= 1
                        IF dependencies[n] == 0
                                queue.push(n)
                        END IF
                END FOR
        END WHILE
        IF solution.length() != nodes.length()
                RETURN ERROR("Problem contained circular dependencies")
        ELSE
                RETURN solution
        END IF
```

You are given a matrix of MxN boolean values representing a board of free (True) or occupied (False) fields. Find the size of the largest square of free fields.

View the answer →Hide answer

A field with a True value represents a 1x1 square on its own. If a field has free fields on its left, top, and top-left, then it's the bottom-right corner of a 2x2 square. If those three fields are all bottom-right corners of a 5x5 square, then their overlap, plus the queried field being free, form a 6x6 square. We can use this logic to solve this problem. That is, we define the relation of size[x, y] = min(size[x-1, y], size[x, y-1], size[x-1, y-1]) + 1 for every free field. For an occupied field, we can set size[x,y] = 0, and it will fit our recursive relation perfectly. Now, size[x, y] represents the largest square for which the field is the bottom-right corner. Tracking the maximum number achieved will give us the answer to our problem.

```
FUNCTION square_size(board)
        M = board.height()
        N = board.width()
        size = Matrix[M + 1, N + 1]
        FOR i IN [0..M]
                size[i, 0] = 0
        END FOR
        FOR i IN [0..N]
                size[0, i] = 0
        END FOR
        answer = 0
        FOR i IN [0..M]
                FOR j IN [0..N]
                        size[i+1, j+1] = max(size[i, j+1], size[i+1, j], size[i, j]) + 1
                        answer = max(answer, size[i+1, j+1])
                END FOR
        END FOR
        RETURN answer
```

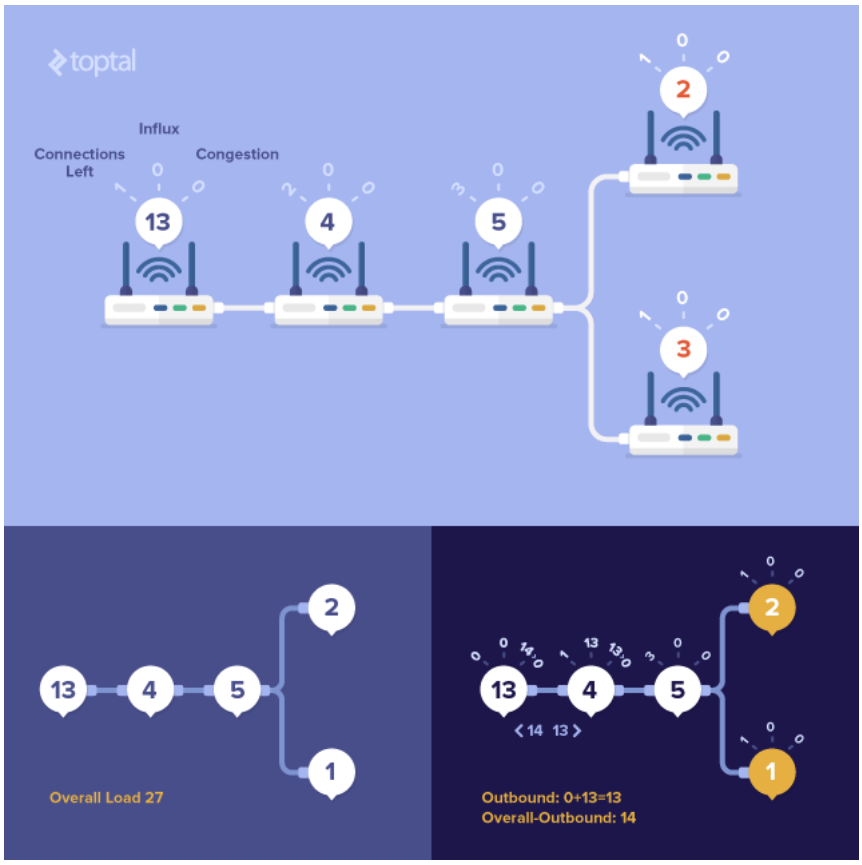
You are given the task of choosing the optimal route to connect a master server to a network of N routers. The routers are connected with the minimum required N-1 wires into a tree structure, and for each router we know the data rate at which devices (that are not routers) that are connected to it will require information. That information requirement represents the load on each router if that router is not chosen to host the master. Determine which router we need to connect the master to in order to minimize congestion along individual lines.

View the answer →Hide answer

First we form an array of connections for each router (`connections` variable). Then we mark the number of connections we haven't processed for each router, and add all routers with one connection to the processing queue. In the tree of routers, these are the leaf nodes. We start from each leaf node and prune the tree along the way, ignoring pruned nodes. Outbound data represents data that the router needs to send to the leftover branch if it doesn't host the master server. Overall load minus outbound data is the data the router will receive from the leftover branch if it hosts the master server. We can then use a for loop to find the leftover branch, add the outbound data to its influx, prune the branch, and check if the other router becomes a leaf node if it is added to the queue. At the end, we retrieve the node with minimum congestion. Throughout the whole algorithm, the `congestion` variable held an array, whereby each element represents the load on the most loaded branch if that router holds the server.

The algorithm runs at O(N) time complexity, which is as efficient as we can get. Pseudocode for the solution is as follows:

```
FUNCTION place_server(loads, router_wires)
        overall_load = sum(loads)
        router_count = loads.length()
        connections = Array[router_count]
        congestion = Array[router_count]
        FOR branch IN router_wires
                connections[branch.first].push(branch.second)
                connections[branch.second].push(branch.first)
        END FOR
        connections_left = Array[router_count]
        influx = Array[router_count]
        queries = Queue
        FOR n IN [0..router_count]
                connections_left[n] = connections[n].length()
                IF connections_left[n] == 1
                        queries.push(n)
                END IF
        END FOR
        WHILE !queries.empty()
                id = queries.pop()
                IF connections_left[n] != 1
                        CONTINUE
                END IF
                connections_left[n] = 0
                outbound_data = influx[id] + loads[id]
                IF overall_load - outbound_data > congestion[id]
                        congestion[id] = overall_load - outbound_data
                END IF
                FOR connection IN connections[id]
                        IF connections_left[connection] == 0
                                CONTINUE
                        END IF
                        Influx[connection] += outbound_data
                        IF outbound_data > congestion[connection]
                                congestion[connection] = outbound_data
                        END IF
                        connections_left[connection] -= 1
                        IF connections_left[connection] == 1
                                queries.push(connection)
                        END IF
                END FOR
        END WHILE
        RETURN minimum_index(congestion)
```



A significantly large static set of string keys has been given, together with data for each of those keys. We need to create a data structure that allows us to update and access that data quickly, with constant time even in worst cases. How can we solve this problem?

View the answer →Hide answer

Let's mark the number of keys as N. The problem presented is a problem of perfect hashing. We do a similar approach as a normal HashTable, but instead of storing collisions in a list, we store them in a secondary HashTable. We choose primary hashing functions until all buckets have a relatively small number of elements in them. After that, in buckets with K keys we place hash tables with K^2 buckets. Even though this seems to result in high memory complexity, expected memory complexity is O(N). By choosing this size of HashTables we have a 50% probability to have collisions. This makes it easy to choose hashing functions that will not result in collisions.

Determine if two rectangles intersect.

1. Give an algorithm to solve this problem when rectangles are defined by their width, height, and (x, y) coordinates of their top-left corners.
2. Give another algorithm where rectangles are defined by their width, height, and (x, y) coordinates of their centers.

What are the behaviors of your algorithms for edge cases?

View the answer →Hide answer

1. You just have to check that one of the rectangle is not completely on the right, left, top or bottom of the other:

```
RETURN !(rect1.x > rect2.x + rect2.width
      || rect1.x  + rect1.width < rect2.x
      || rect1.y > rect2.y + rect2.height
      || rect1.y + rect1.height < rect2.y);
```

2. You calculate the distance in x and y between both centers and compare it to half of the sum of their width and height, respectively:

```
RETURN abs(rect1.x - rect2.x) <= (rect1.height + rect2.height) / 2;
```

The only edge case is if one of the rectangles is completely included inside the other one, as the word “intersect” is not necessarily very clear about the behavior to have in this case. Both algorithms consider this to be an intersection, so if that’s not what’s needed, it will take some extra checks to remove this case.

You have a set of date intervals represented by StartDate and EndDate. How would you efficiently calculate the longest timespan covered by them?

What will be the time complexity?

View the answer →Hide answer

```
FUNCTION MaxTimespan(StartDates, EndDates)
   Sort(StartDates, EndDates)
   ActStartDate = StartDates[1]
   ActEndDate = EndDates[1]
   ActTimespan = ActEndDate - ActStartDate
   FOR i in 2..StartDates.Length
      IF StartDates[i] BETWEEN ActStartDate AND ActEndDate
         ActEndDate = MAX(ActEndDate, EndDates[i])
         ActTimespan = MAX(ActTimespan, ActEndDate - ActStartDate)
      ELSE
         ActStartDate = StartDates[i]
         ActEndDate = EndDates[i]
      END IF
   END FOR
   RETURN ActTimespan
```

The overall time complexity is O(NlogN) because:

1. Sort intervals by start date. Time complexity is O(NlogN).

2. Take first interval as actual range. Loop over intervals and if the current StartDate is within the actual range, extend EndDate of the actual range if needed and extend maximal timespan achieved so far if needed. Otherwise, use current interval as new actual range. Time complexity is O(N).

* There is more to interviewing than tricky technical questions, so these are intended merely as a guide. Not every “A” candidate worth hiring will be able to answer them all, nor does answering them all guarantee an “A” candidate. At the end of the day, hiring remains an art, a science — and a lot of work.