DIVIDE AND CONQUER

https://www.toptal.com/algorithms/interview-questions
https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/

1. Divide — split into subproblems of same type
2. Conquer — recursively solve these subproblems
3. Combine — appropriately combine the answers

Examples:

1. Binary Search
2. Quicksort
3. Merge sort
4. Closest pair of points $O(n \log n)$
5. Strassen's algorithm - matrix multiplication
6. Cooley-Tukey FFT $O(n \log n)$
7. Karatsuba algorithm - fast multiplication

CALCULATE p^k

How do you optimally calculate when k is non-negative integer? What is complexity of the solution?

$p^k = p^x \times p^y$ where $x+y=k$

$(2^3 = 2^1 \times 2^2)$

For even value of k: a = k b = k/2

thus p^k = (p^2)^(k/2)

For odd value of k: x = 1 y = k-1 results in y being even

```
def pow(base, exponent):
    if exponent == 0:
        return 1
    elif exponent % 2 == 0:        # EVEN
        return pow(base*base, exponent/2)
    else:                          # ODD
        return base * pow(base*base, (exponent-1)/2)
```

INSERTION SORT, HEAP SORT, MERGE SORT, QUICKSORT

Explain how each of these work (and complexity)

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

Advantages of each sort AND time/memory complexity?

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.

HASH TABLE

What is it? What are average/worst-case times for each of its operations? How to use it to find anagrams in a dictionary?

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).

ARRAY NUMBERS AND OPPOSITES

Array of length N. Find all positive numbers that have their opposite in array as well. Describe approaches for optimal worst-case and optimal average-case solutions.

Solution 1:

- custom comparison that places negatives just before their positive counterparts
- O(n log n) sort (this is the bottleneck)
- find values where $A[n-1] = -A[n]$

Solution 2:

- hash on key = abs(A[n]), value = A[n]
- if key exists in table T:

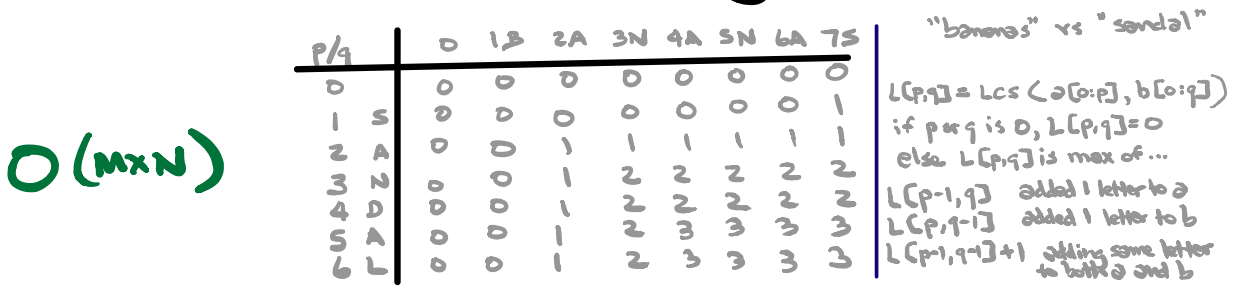
```
if T[key] == -A[n]:
    A[n] is in solution
    T[key] = 0
else:
    T[key] = A[n]
```

- complexity O(N) because hashtable ops are O(1) performed N times

DYNAMIC PROGRAMMING

longest common subsequence

Describe dynamic programming. Use it to find LCS in 2 arrays.



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".

(see above 2D matrix)

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)

RED/BLACK TREES vs B-TREES

- both are balanced trees
- red/black are balanced binary trees where longest branch is never more than 2x as long as shortest branch (one extra bit on each node)
- B-Tree can have more than 2 children for each node

Red–black tree

Type	tree
Invented	1972
Invented by	Rudolf Bayer

Time complexity in big O notation

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$[1]	$O(\log n)$[1]
Insert	$O(\log n)$[1]	$O(\log n)$[1]
Delete	$O(\log n)$[1]	$O(\log n)$[1]

DIJKSTRA AND PRIM ALGORITHM

What are they, and how are they implemented?
How does Fibonacci Heap relate to them?

Dijkstra = single source shortest path
Prim = minimum spanning tree
Both require a priority queue in their implementation,
and Fib Heap is better than Binary Heap or Binomial Heap.

Fibonacci Heap: find-minimum, insert, decrease key = O(1) time
delete = O(log n) time
Using Fibonacci Heap reduces running time for Dijkstra's alg.

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).

BELLMAN-FORD ALGORITHM VS DIJKSTRA

Bellman-Ford is slower than Dijkstra, but it is more versatile because it is capable of handling a graph where some edge weights are negative.

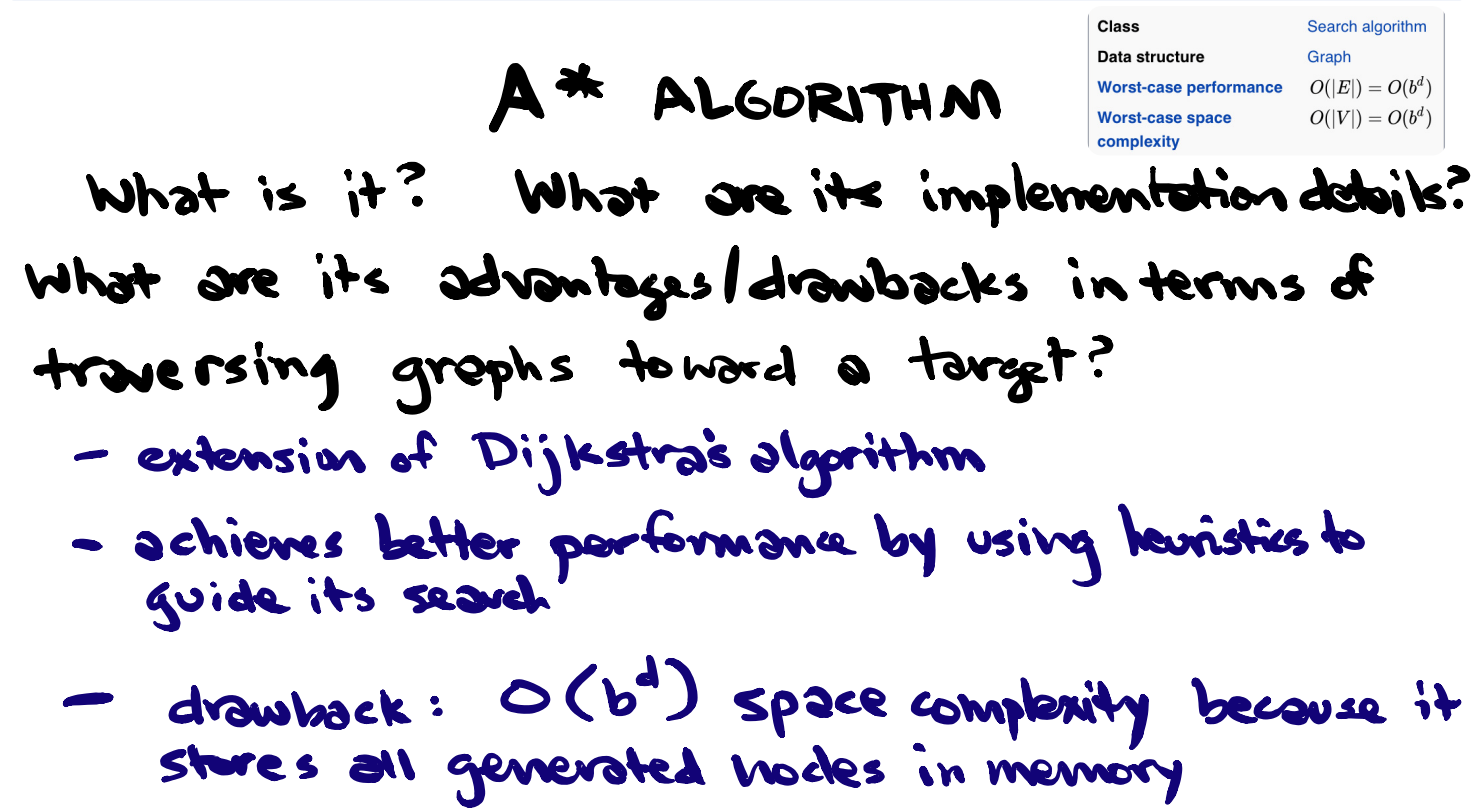
The Bellman-Ford algorithm finds single source shortest paths by repeatedly relaxing distances until there are no more distances to relax. Relaxing distances is done by checking if an intermediate point provides a better path than the currently chosen path. After a number of iterations that is slightly 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.



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.

SUM OF SUBARRAY

Given 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 between sum queries — what is optimal solution then?
- What's the preprocessing and query complexity for each solution?

example

i: 0 1 2 3 4 5 6 7

A [1 2 5 4 3 7 2 1]

sum: 1 3 8 12 15 22 24 25

we can preprocess to create a sum array (also size N)

subarray-sum [i1 = 5 to i2 = 7] = sum[7] - sum[4]

= sum[i2] - sum[i1-1]

preprocessing: O(n)
query: O(1)

This solution breaks, of course, if we update the array (because our sum array would no longer be valid).

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.

TASK SCHEDULER

Design a scheduler to schedule a set of tasks.

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

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.

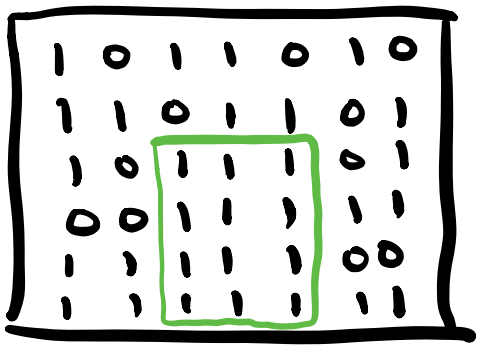
After the algorithm executes, if the list doesn't have as many elements as the number of tasks, we have circular dependencies. Otherwise, we have a solution:

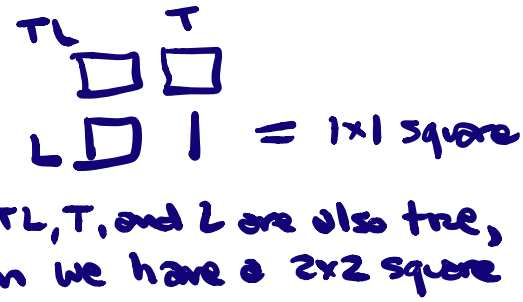
```
FUNCTION schedule(nodes)
    dependencies = Array[nodes.length()]
    FOR node IN nodes
        FOR child IN node.children
            dependencies[child] += 1
        END FOR
    END FOR
    queue = Queue
    solution = List
    FOR n IN [0..nodes.length()]
        IF dependencies[n] == 0
            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
```

LARGEST SQUARE IN BOOLEAN MATRIX

Given M×N matrix of boolean values (where Free is True and Occupied is False).

Find the size of the largest square of free fields.





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
```

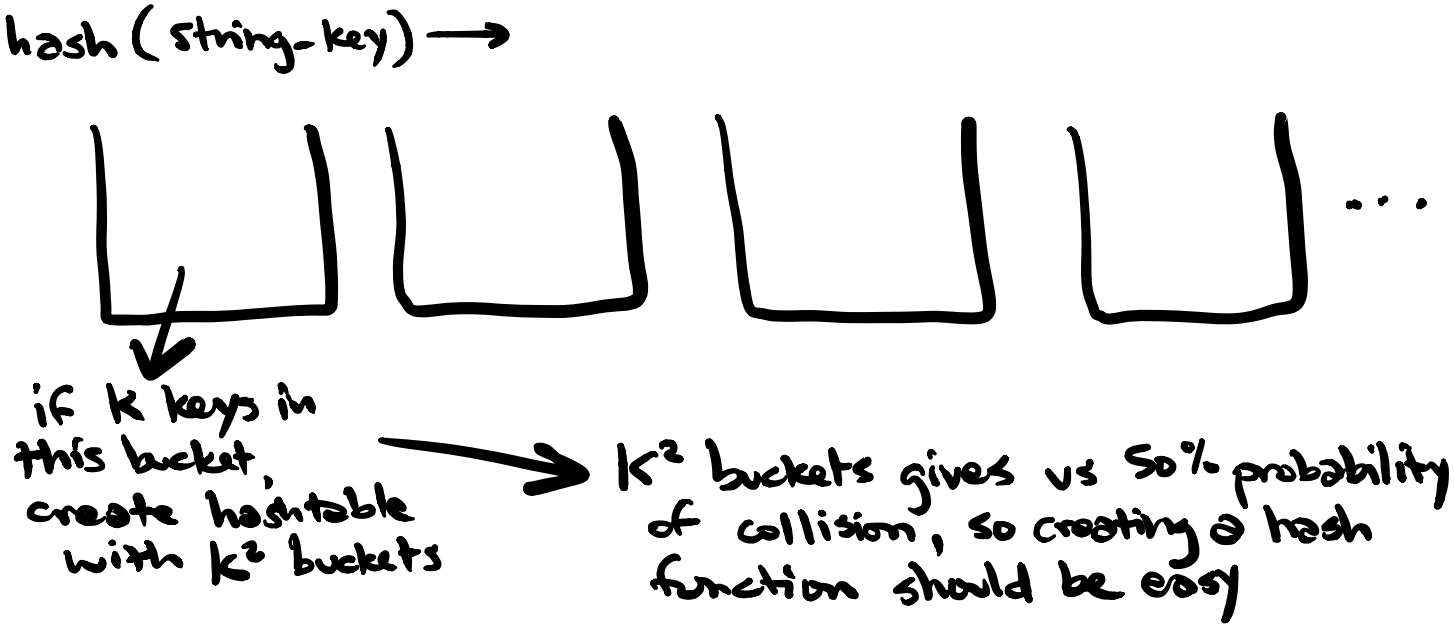
LARGE STATIC SET OF STRING KEYS

A significantly large static set of string key is given along with data for each key.
We need to create a data structure that allows

us to update and access that data quickly (in constant time worst case).

How can we solve this problem?

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.



INTERSECTING RECTANGLES

Determine if two rectangles intersect.

case 1: given width, height, and (x,y) coordinates of upper-left

case 2: given width, height, and (x,y) coordinates of center point

What are the behaviors of your algorithms for edge cases?

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.

LONGEST TIMESPAN

Given a set of date intervals: (start date, end date)

How can we efficiently calculate the longest timespan covered by them? (And what's the complexity?)

TRAVERSE N METERS WITH 1m,2m,3m,4m,5m JUMPS

Design algo to find #of ways to traverse N meters using jumps of 1m, 2m, 3m, 4m, 5m.
What is complexity?

K can be reached from k-5, k-4, k-3, k-2, k-1

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

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.

OPTIMAL NETWORK ROUTE

Task: choose the optimal route to connect a master server to a network of N routers.

The routers are connected with the minimum N-1 wires in a tree structure.

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 to which router we should connect the master in order to minimize congestion along individual lines.