CPS 305 Data Structures, Weekly Labs

Lab 1: A Linear Mind

Zero. For the search pattern `"ABBAABBAA"`, compute by hand the Boyer-Moore *good suffix table*, the DFA *transition table*, and the Knuth-Morris-Pratt *failure table*.

After that first problem, perhaps better suited for a mindless mechanical machine than man, it's time to harness the human mind and your creative powers of imagination. The following problems should at this point be trivial for anybody who is still with us to solve in $O(n^2)$ time using two nested for-loops. However, the challenge is to make all these run in guaranteed $O(n)$ worst case time by not being a "Shlemiel" who tires himself running back and forth for no good reason! Remember also that $O(n)$ is not the same thing as $O(nk)$, unless k is fixed to some constant value so that it vanishes from the O-notation as a constant factor.

One. Given an n-element array of integers (that can be either positive or negative) and an integer k, find the k-element contiguous subarray with maximum sum of values.

Two. An n-element integer array is a *jolly jumper* if the absolute differences of its successive elements contain all possible values in 1, ..., n - 1. For example, the five-element array 6, 3, 1, 2, 6 is a jolly jumper, since the absolute differences of its successive elements are 3, 2, 1, 4. Determine whether the given array is a jolly jumper. (Don't ask me why such an array is called that. This problem is taken from some European programming exercise site where they called it that.)

Three. Given an n-element array that contains **positive integers only** (so there are no negative elements anywhere), and a positive goal value k, find the length of the longest contiguous subarray whose sum of elements equals exactly k. (Hint: use the technique of two chasing indices to delimit that subarray, moving one of these indices forward depending on what is going on in the subarray between them.) If no such subarray exists, this method should return -1.

Four. Determine whether the given n-element array contains a *strict majority*, that is, *more than half* of its elements are equal to each other. (This definition is intentionally phrased as "more than half" rather than "at least half" to ensure that the strict majority value is unique whenever it exists.)

This problem seems literally impossible at first since it seems to inherently require at least $O(n \log n)$ comparisons to solve in the worst case, achieved by sorting the array and then looping once through it. To get started, note that skipping over $2k$ elements for any $k > 0$ from the beginning of the array so that exactly k of those elements are equal to each other whereas the other k are some different values can never change the strict majority of the remaining array, assuming that the strict majority exists in the first place. (Example: if 42 is the strict majority element of the array, after

skipping over ten elements that are equal to 42 and ten other elements that are anything else but 42, the element 42 will still be the strict majority element of the remaining array.)

Five. The suffix array of the given string is an integer array of the same length that contains the positions of all its nonempty suffixes sorted in alphabetical order. For example, the suffix array for the string `"ilkka"` would be `[4, 0, 3, 2, 1]`, since these suffixes listed in alphabetical order would be `"a"`, `"ilkka"`, `"ka"`, `"kka"` and `"lkka"`. Once the suffix array of the given string has been constructed, how would you use it to quickly determine whether that string contains the given search pattern? What is the asymptotic running time of your substring search algorithm?

Lab 2: Heaping Servings of Tasty Goodness

One. The wordlist file `sgb-words.txt` that is used in the last two programming projects of this course contains a subset of 5757 five-letter words of English, as chosen and set in stone by none other than Donald E. Knuth, sorted by their estimated frequency in actual use. Why is this order more useful than listing these words in alphabetical order? In this same spirit, is it better for an array search algorithm to return the position of the key that was found, or simply the key that was found, or maybe just a boolean truth value telling whether the key was found?

Two. Devise an algorithm to find the *second largest* element of an n-element array, using only pairwise element order comparisons. Assume that all elements in the array are distinct. What is the *exact* number of element comparisons that your algorithm has to perform in the worst case? Putting on your adversary hat, create the array that causes this worst case scenario.

Three. Display the contents of the array when the following elements are inserted to an initially empty *binary min-heap* in this order: 6, 2, 8, 5, 1, 3, 9, 4. Then display the contents of the array in each stage of calling `Extract-Min` eight times.

Four. Given two binary heaps A and B that have n and m elements, respectively, what is the fastest way to combine these two into a new $n + m$ element binary heap C that contains those same elements? What is the asymptotic running time of your procedure?

Five. Given a binary max-heap with n elements, what is the fastest way to convert it into a min-heap with the same elements? What is the asymptotic running time of your procedure?

Six. A d-ary heap is a straightforward generalization of binary heap so that each node has d children, and the operations are the obvious generalizations of those of the binary heap that we get as special case of d-ary heap simply by fixing $d = 2$. How does this change affect the formulas for `Parent(i)` and `Left(i)` ? What is the asymptotic running time of the priority queue operations `Push` and `Extract-Max` ?

Seven. Given n separate binary heaps that contain up to m elements each, what is the asymptotically fastest way to combine these elements into one binary heap that contains all these elements? Suppose n and m are both somewhere in the high thousands. Analyze the asymptotic running time of your algorithm. What if you wanted to create a binary heap that contains only the k smallest elements of all these heaps together, with k being much smaller than nm?

Lab 3: Linked Data Structures

One. Design a data structure that operates as an otherwise ordinary LIFO stack but with the additional operation `Find-Min` that returns (but does not remove) the current smallest element in that stack. All three operations `Push`, `Pop` and `Find-Min` should still work in O(1) worst case time.

This exercise is the first of many similar problems in your life as a computer scientist in which you have to **augment some data structure** with additional data so that it can still perform its original operations efficiently but also gives you new capabilities that your current application happens to need. For this reason, computer scientists need to understand how data structures work under the hood so that they can implement them when the standard library version does not have these augmented capabilities, seeing that you don't get to access, let alone, edit the source code of `LinkedList<E>` and others, and using inheritance to create your custom version of that data structure does not help here since it does not grant you the needed access to the `private` data of the superclass.

Two. Suppose that you are given the start node of a singly linked chain of nodes that is supposed to eventually end with some `next` reference that has the value `null`. However, some sneaky little bugger might have corrupted this chain so that the `next` reference of some node in it loops back to some earlier node in that same chain, which would cause the chain traversal loop to never reach the `null` reference that ought to be in the last node to stop this traversal. Devise an efficient algorithm to check whether the chain hanging from the given start node is a proper chain so that it does not contain a loop back to some earlier node.

Three. Suppose that we know that inside an `Extract-Min` priority queue of integers that is *monotonic* (that is, no elements are ever inserted that are smaller than the currently smallest element at the head of the priority queue), the difference between the smallest and largest key that can exist inside this queue at the same time is known to be at most k. (Neither of these is true for a binary heap in which you can always push any arbitrary integer value regardless of what other values are already stored in it.) Design a priority queue data structure that allows `Push` and `Extract-Min` to work in guaranteed O(k) worst case time completely independent of n, the number of keys currently stored in the queue.

(In a *discrete simulation* application, this priority queue contains the future *events* that are guaranteed to happen but waiting to be processed. Processing the next event at the head of this

queue can cause some number of new future events to become inevitable and thus added to this queue. If the temporal rules of causality inside that simulated world do not allow any event to cause another event to happen in the past, monotonicity of the queue automatically follows. If the causality to the future also has a fixed *horizon limit* of length k up to how far in the future some event could theoretically cause new future events to happen, that horizon k automatically becomes the maximum possible difference between the earliest and the latest events inside the simulation queue.)

Four. Suppose you are given two LIFO stacks as *black boxes* so that you can freely use their operations `Push` and `Pop`, both guaranteed to be O(1) in the worst case, but you cannot otherwise access or modify the internal data storage of these stack objects in any way. Show how you can still implement a FIFO queue using only the public operations of these two stacks and O(1) other data storage. Your FIFO queue `Push` and `Pop` operations should work in amortized O(1) time.

Five. The defining characteristic of *imperative programming* is the *destructive assignment* so that whenever some variable or a larger data structure is modified, the previous version of that variable or data structure no longer has any physical existence. Only the present moment really exists with no past history, from which the next program statement that is executed destroys the old present moment and creates the next present moment.

But sometimes we would also like to remember the past so that we can return to it if needed. A data structure is said to be persistent if every mutative operation performed on it always creates a new object to represent the resulting new *version* of that data structure, so that the object representing the previous version still continues to exist in memory, fully accessible from any references that you still have pointing to it. (Note that this meaning of "persistent" is different from the usual meaning of this word in computer science, that is, *data that outlives the process that created it*, such as files and databases.) So that each new version of the data structure would not require copying all the elements from the previous version and thus be inherently at least O(n) in both time and space, the new version should share as much of the data of the previous version as possible. (Since objects are always immutable under this scheme, we also get all the benefits of immutability for free. Nothing to sneeze at!)

All right, then. After that mouthful, design a persistent data structure to represent a LIFO stack, with guaranteed O(1) worst case operations.

Six. Design an efficient data structure that works as priority queue of *calendar events* of the form `(day, month, year, event)`. Your queue should support the operations `Extract-Min` to remove the next event from the queue, and `Push` to add a new event into the queue. Make both operations work in almost certain O(1) time with the assumption of *monotonicity* so that once some event has been extracted from the queue, no earlier events will ever again be pushed into the queue, and furthermore, that the overwhelming majority of events that are pushed in will take place within a couple of days of the current earliest element in the queue. Unlike the data structure of the previous question three, your data structure should not have any upper limit k for the

horizon, but must allow pushing in events that will take place in arbitrarily far in the future. For example, the future event `(1, 1, 9595, "Take everything this old Earth can give")` could be pushed in today so that this event would event-ually (heh, these dad jokes just keep writing themselves) emerge from your queue once all the events in between today's date and the date of January 1 of the year 9595 have first been popped and processed.

Lab 4: Comparison Sorting

One. Given an unsorted array that contains n different elements, your task is to find and list its k largest elements in sorted order using only element order comparisons. Devise an algorithm to do this in the asymptotically optimal fashion.

Two. Unlike merge sort, quicksort is not *stable*. How would you turn any non-stable comparison sorting algorithm stable? How much additional time and space does your scheme require?

In the same spirit, how can you make any sorting algorithm to run in $O(n)$ best case time? (Of all the comparison sorting algorithms that we have seen so far, only insertion sort has the $O(n)$ best case scenario right out of the box, for arrays that are "almost sorted". Quicksort and mergesort are still $O(n \log n)$ even in the best case, even if a *benevolent genie*, the opposite of the adversary who we have faced so far, gets to set up the array elements so that the algorithm takes as little time as possible.)

Three. Suppose that it is guaranteed that every element of an n-element array is at most k steps away from the position that it would end up in the sorted array. If you are told the value of this k, how could you sort the array in guaranteed $O(n \log k)$ time? ("Just use insertion sort" is not the correct answer here, since its running time is $O(nk)$.) What if you were not told the actual value of k beforehand, but were only given a gentleman's guarantee that some such k exists?

Four. As the sweet sounds of The Doobie Brothers and The Commodores from the tinny transistor radio fill your small smoke-filled office whose walls are mostly some tacky shade of brown or yellow and hairy furniture, you slowly come to accept that some mysterious force has whisked you back in time to the seventies to set right what once went wrong. Your boss proudly presents to you the new minicomputer purchased by your company (in the spirit of the era, it comes with wood-panelled sides adorned with flower stickers) that comes with a whopping 128 **kilo**bytes of RAM, but its secondary storage can store a whopping 256 **mega**bytes of files. Of these 128 kilobytes, your program can use up to only 16 kilobytes at any given time. Files can be read and written sequentially one line of text at the time, and several file handles can be kept open at the same time to interleave the reading and writing of those files.

After the day is over with the serious work for The Man, it's time to play some huge text adventure games... but before you get to do that, copied from a tape secondary storage into this hard drive there is an unimaginably huge 32 megabyte text file whose each line contains a customer record

(separated into fields with commas) that the company has slowly accumulated over its history of operations. Your boss assigns you the task of creating a new version of this file that contains these same records in sorted order based on one particular field. How would you write a program to achieve this within the confines of your very limited 16 kilobytes of process RAM? (Programmers routinely had to write their own sort and other routines until about the mid-nineties.)

Five. Having solved the previous problem well enough so that the mysterious force is fully satisfied and brought you back to the present day, you see Joe Palooka barreling into your present day office all excited. Joe gushes how he will now surely become famous because he has invented a new comparison-based priority queue data structure that allows both operations `Push` and `Extract-Max` to work in O(1) amortized time. Explain why Joe is sadly mistaken, in the same curt spirit as Joe's physics professor would reply if Joe ever similarly came to his office gushing about a new kind of a perpetual motion machine that he had invented.

Six. The fastest way to sort a very small array `A` whose size is fixed to a handful of elements is to simply hardcode the comparisons and swaps into the program statements without using any loops at all. Assume that the operation `CEX(i, j)` **C**ompares the array elements `A[i]` and `A[j]` and **EX**changes these elements if they are in the wrong order.

(a) For the values of n from 2 to 4, create the shortest possible sequences of `CEX`-operations that is guaranteed to sort any given n-element array.
(b) A *sorting network* consists of a feedforward structure of `CEX`-operations so that two or more consecutive `CEX`-operations that do not access the same elements can be executed in parallel. For the values of n from 2 to 4, create the fastest possible sorting networks of `CEX`-operations.
(c) Strange but true: adding a `CEX`-operation into a correctly working sorting network can cause that network to no longer correctly sort some particular input permutation! Demonstrate this by starting with a sorting network that works correctly for all inputs of $n = 4$, and add one more innocent looking `CEX`-operation to your network so that the resulting network no longer correctly sorts every possible input permutation.

Seven. You are given n nuts and n bolts, all initially unsorted. All nuts and bolts are different sizes, and each nut fits exactly one bolt. It takes you O(1) time to try to fit a nut with a bolt, this operation always ending with one of the three possible results "the nut is too small", "the nut and bolt fit together just right" and "the nut is too large". Devise an algorithm to correctly match all nuts and bolts with an $O(n \log n)$ average case running time. (The sizes of nuts and bolts are too close to each other for you to be able to eyeball any sort of bucketing or similar rough partitions as a preprocessing stage to speed up the actual matching stage of individual nuts and bolts.)

Lab 5: Unconventional Sorting

One. Sort the following strings using LSD radix sort: `BOB`, `TAP`, `TOP`, `BAT`, `COT`, `TAB`, `CAT`. Show the order of these strings after each pass of the algorithm. Repeat this exercise using MSD radix sort.

Two. You need to sort an array of n floating point numbers, but this time these numbers are drawn randomly from some other continuous probability distribution than the simple uniform distribution. If you know the *cumulative probability distribution* function (CDF) of that probability distribution, how would you modify the basic *bucket sort* algorithm to sort these numbers in the $O(n)$ average case time?

Three. We often need to sort physical things inside this big old physical universe of ours that, the stingy miser that it is, does not grant us O(1) random access to its arbitrary positions. For each of the following physical objects, explain how you would sort them most efficiently using only your brain and two hands. Analyze the asymptotic running time of your "hands-on" algorithms for the general n. Explain why your technique works for these physical objects better in the physical world than quicksort or some other algorithm designed for computer processors that operate under the luxury of the RAM computation model.

1. An ordinary deck of 52 playing cards, to be sorted by suit and rank.
2. A stack of 500 unsorted Ryerson exam booklets, to be sorted by student last and first name.
3. A pile of metal rods of uniform thickness, to be sorted by their length.

Four. Tukey's ninthers, mentioned in the Princeton slides as a method to quickly approximate the median of nine elements, is a simple but highly robust way to approximate the true median of an entire array of arbitrary length in guaranteed linear time. Implement this method so that it recursively solves the problem for the three element subarrays. Then use random sampling of all possible permutations of an array to estimate how often this algorithm produces each possible estimate for the median.

Five. In the problem of **0-1 sorting**, there are only two possible values for keys so that each key is known to be either zero or one. (These elements can still have additional satellite data fields to distinguish them from each other.) How would you sort these elements in linear time using O(1) extra memory? Is your sorting algorithm stable?

Six. Suppose that your computer has c idealized processor cores that operate perfectly in parallel while sharing the same RAM memory. As long as these processor cores don't try to read or write the same location in the array, everything will be fine. (Otherwise, we need to enforce *mutual exclusion* to shared data with expensive *concurrency locks* that will slow down the execution and might even eat away all of the speed gains from parallelism.) Considering the now familiar sorting algorithms selection sort, insertion sort, merge sort, quicksort, heapsort, counting sort and radix sort, sort (heh) these algorithms in an approximate order of how easy it would be to speed up their operation with parallelization.

Lab 6: Hashing

One. In a hash table where collisions are resolved with chaining, how would keeping the elements of each chain in sorted order affect the asymptotic running time of the three basic dynamic set operations?

Two. In a hash table where collisions are resolved with chaining, we can easily keep track of the total number of elements n, the number of slots m, and the length of the currently longest chain L. How would you efficiently choose a random element from this table so that each element stored in the table has the exact same uniform probability $1 / n$ of being selected? What is the worst case running time of your random choice algorithm?

(The simplistic algorithm "choose a random slot until you find some slot that is not empty, then choose a random element uniformly from the chain that is hanging from that slot" most empathetically does *not* produce the required uniform distribution. Why is that?)

Three. How would you modify the chained hash table to guarantee that iterating through the keys currently stored in the tree can be done in $O(n)$ time with the guarantee that the iteration will go through the keys in the exact same order in which these elements were originally added to the table? How much additional time and space does your modification entail? The data structure must still support arbitrary dynamic set operations, but for simplicity, you can assume that the structure is not modified during the iteration stage.

Four. Tabulation hashing is a technique to hash a k-bit integer into an m-bit hashcode. It uses a k-element array A of m-bit integers generated randomly at the startup of the program. The hash value of the given integer n is the **exclusive or** of those positions of A for which the integer n has a bit turned on. Computing the hash value for an integer n will therefore take $O(k)$ time, but the main advantage of this technique is that when n changes slightly, so that only one or two bits inside it flip and the rest of the bits remain as they were, the new hash value can be computed very quickly. Also, computation can use more than one bit at the time.

Zobrist hashing is a clever way to use tabulation hashing to compute the hash value for a position in a board game. Suppose we are writing an AI player for some board game where each of the 64 squares of the game board can be either empty, have a white piece, or have a black piece. The minimax search algorithm to actually evaluate the moves will be explained in CPS 721 *Artificial Intelligence*, but we can already implement its *transposition table* to speed up the minimax search.

A transposition table is a hash table that stores the game positions already encountered during minimax search, along with their associated values, so that if we encounter these same states again along some other path during the search, we don't need to compute those values again from scratch. How big does k need to be to encode the given position into a k-bit integer?

Suppose then that we have already computed the k-bit hash value h_1 for the current position. One black piece moves from square s_1 into another square s_2 that previously contained a white piece that is captured and removed from the game, replaced by the black piece moving into that square. How would you quickly compute the hash value h_2 for the new position that results from this capturing move?

Five. An *overflow table of order k* is a cutesy old variation of the open addressing hash table that consists of an array of 2^{k+1} elements so that the first 2^k elements form its *primary area*, and the remaining 2^k elements for an overflow table of order k - 1. Each order k uses its own randomly generated hash function h_k. To insert a key into an overflow table of order k, first try to insert that key in its primary area. If there is a collision, continue inserting the key to the next overflow table of order k - 1. If this insertion fails because of a collision in the smallest available overflow table, double the size of the entire table and rebuild the whole thing from scratch, similarly to the way that cuckoo hashing takes care of the same problem by throwing memory at it. Explain how the search and remove algorithms would work for an overflow table of the order k, and analyze their asymptotic running time.

Lab 7: Dynamic Programming

One. In mergesort and quicksort, each recursive call generates two more calls. So how come the running times of these algorithms are not exponential? Also, why can't these algorithms be sped up by use of memoization? In the same vein, why can't the exponential recursion of Towers of Hanoi be sped up with the use of memoization?

Two. A tree structure consisting of nodes contains a positive integer key in each node. Your task is to find a subset of nodes to maximize the sum of its keys, under the constraint that no node and its parent node can be simultaneously chosen to this subset. Design an efficient dynamic programming algorithm to find the subset of nodes that maximizes this sum. As usual, start by writing the recursive equation $M(s)$ for the maximum sum that can be constructed from the tree with the root node s. For each node, you will once again either "take it" or "leave it", whichever way leads you to the optimal solution.

Three. Given two strings A and B of lengths n and m, and a third string C of length $n + m$, determine whether C can be constructed by somehow interleaving all of the characters of A and B without changing the order of these characters inside the original string. For example, "**he*w*l*o*r*l**lo*d***" or "***w*h*o*e*r*ll*l*d*o***" could be interleaved from "**hello**" and "***world***". First you should realize that the simplistic greedy choice algorithm "Find all the characters of A inside C in order, then check that the remaining characters of C form B" does not work, as can be seen from the string "**hell*world*o**" that this algorithm would erroneously reject.

Four. The *subset sum problem* consists of a sorted array of positive integers and a goal value g, and your task is to determine whether there exists some subset of array elements whose sum is equal to g. You can use each element at most once. For example, given the array [3, 5, 8, 9, 11, 14, 16] and the goal value 31, one possible solution would be [3, 8, 9, 11]. Design an efficient dynamic programming algorithm to solve this problem in $O(ng)$ worst case time and space. As usual, start by writing the recursive equation for $S(n, g)$ of whether there exists a subset of first n elements of the array whose elements sum up to g. How would you modify your algorithm to also return the required subset instead of simple yes-no answer for its existence?

Five. Your task is to fill an n-by-1 box using any combination of *black tiles* of size 1-by-1 and *red tiles* of size k-by-1 where you get to freely choose k as long as it is at least 3. However, no red tiles can be placed next to each other, whereas black tiles may be placed next to both black and red tiles. Design an efficient dynamic programming algorithm to count the number of different ways to fill an n-by-1 box. (For an illustrative picture of this, this is problem 114 in Project Euler.) As usual, start out by writing the recursive equation $W(n)$ for the number of different ways to fill in an n-by-1 box.

Lab 8: Binary Search Trees

One. Show the result of inserting the elements 7, 2, 6, 1, 9, 8 and 5 into an initially empty binary search tree in this order. Then show the result of removing the key in the root node of the tree using the Hibbard deletion.

Two. What is the best way to convert an already sorted n-element array into a balanced binary search tree that contains those elements as its keys? What is the asymptotic running time of your method?

Three. Generalize the binary search tree key searching algorithm to implement *range search*, that is, instead of looking for only one particular element, this function gets two parameters *min* and *max*, and visits all nodes whose values are between *min* and *max*, inclusive. Unlike the basic search algorithm to find one key, the range search algorithm might have to continue recursively to both children of the current node instead of just one of them. Be careful about the condition that determines which children you continue into; this condition should *depend only on the key inside the current node*, but not on the keys of either child, as those nodes might be NULL anyway! Verify also that when *min* = *max*, this range search algorithm degenerates into ordinary BST search algorithm searching for the single key *min*.

Four. A simple and fun way to encode arbitrary binary tree structures (that is, only the structure, not the keys in the nodes) one-to-one into nonnegative integers using *bitwise arithmetic* techniques works by first defining the base case that says that the integer zero corresponds to an empty tree. Any other integer $n > 0$ corresponds to the binary tree that has a root node and left and right subtrees encoded by numbers n_1 and n_2 where n_1 is constructed from the bits of the even positions

of the binary representation of n - 1, and n_2 is similarly constructed from the bits of the odd positions of the binary representation of n - 1. (Why n - 1 instead of n?)

(a) What value of n corresponds to the binary tree structure of size three that has root a node and both left and right child nodes?
(b) What binary tree structure encodes to the number 17?

A random binary tree structure can now be chosen uniformly over all possible binary tree structures simply by first creating a random integer and converting it into a binary tree. Alternatively, we can easily loop through all possible binary tree structures.

Five. The recursive tree walk goes through the nodes of binary tree in O(n) time regardless of the shape of the tree. How would you modify this algorithm to verify that the keys in the tree satisfy the BST property throughout? The tree walk must still work in O(n) worst case time, so this cannot be done in "Shlemiel" style by making two separate tree walks starting from each node to verify that all keys in the left subtree are smaller than the current key, and that all keys in the right subtree are larger than the current key. Instead, you need to do all checking in one pass through the tree that spends constant time in each node.

Six. *Fibonacci trees* are a historical name for AVL trees. The name is appropriate in that any AVL tree whose height is h must contain at least F_h nodes, where F_h is the h:th Fibonacci number. Since there ought to be at least one mathematical proof in this course, prove that claim by induction on the height of the tree. Since Fibonacci numbers grow exponentially with respect to h, the height of the tree conversely grows logarithmically with respect to n.

Lab 9: Graphs

One. The **state spaces** of various **puzzles** (that is, one-player games) can be expressed as graphs whose vertices correspond to the states s that the abstract puzzle can be in, and whose directed edges connect the vertices (s_1, s_2) if it is possible for the puzzle to move from state s_1 to state s_2 in one move. For the puzzles whose each move is reversible, such as Rubik's cube, this state space graph can then be considered undirected. When talking about state spaces, we often talk directly about "states" and "transitions" of the graph instead of "vertices" and "edges". To-may-to, to-mah-to.

Let us now consider the famous 15-puzzle played by moving 15 tiles inside a 4-by-4 board, trying to reach the goal state in which these tiles are in sorted order.

(a) How many different states does this puzzle have?
(b) How many transitions are there emanating from each state?
(c) Suppose we wish to solve this puzzle with the A* search algorithm, as so many second- and third-year students have done before us as their programming projects all around the

world. Devise a heuristic function h(s) that never overestimates the number of moves needed to reach the goal state from the current state s. Try to make your heuristic as tight as possible, to guide the expansions performed the A* algorithm to expand as few nodes as possible along the way. Of course, your heuristic function also needs to be fast to compute so that the time spent in its computation does not eat away the savings in expanding fewer nodes with the aid of a better heuristic.

Two. Depth first search algorithm can be augmented to solve many other interesting graph properties, such as **topological sorting**. However, for the particular problem of topological sorting we already have a much simpler algorithm. Repeatedly find a node that has no incoming edges. Give that node the next running number, and remove the node and its outgoing edges from the graph. Repeat this until all nodes have been numbered. How would you implement this algorithm to work in O(V + E) guaranteed time, and so that "remove the node" operation is merely conceptual so that the actual graph data structure is not modified in any way during this topological sort? What would happen if you executed this algorithm in a graph that is not a DAG but contains directed cycles?

Three. Prove that if every edge of a connected graph has a unique length, the MST of that graph is also unique. Start by assuming the counterexample, that there exist two different MST's, lets call them T_1 and T_2, with the equal total length. Take the shortest edge e that is in one MST but not in the other, and derive a contradiction from what would happen if the edge e were added to the other MST. (Why does this proof not lead to a contradiction unless all edge lengths are unique?)

Four. What happens to the minimum spanning tree if the same constant value c is added to the length of every edge? Corollary 1: Analogous to Dijkstra's algorithm, does the MST calculation also require that the edge lengths are nonnegative? Corollary 2: Suppose you wanted to find the *maximum spanning tree*, a spanning tree of V - 1 edges whose total weight is the *largest* possible. How would you modify the MST algorithm to achieve this goal?

Five. (a) Kruskal's algorithm for generating the minimum spanning tree can also be executed "in reverse" by starting with the complete graph with all the edges included. Repeatedly find some cycle, doesn't even matter which, and remove its longest edge. Keep doing this until exactly V - 1 edges remain. What is the asymptotic worst case running time of this algorithm? (b) Another variation of Kruskal is the so-called Reverse-Delete algorithm. Iterate through the edges in *descending* order of length, and for each edge, check whether its removal would disconnect the graph. What is the asymptotic running time of this algorithm?

Six. Breadth-first search (or Dijkstra's algorithm, when your edges can have different lengths) can find the shortest path from one particular start node into whichever is the nearest one from the set of designated *goal nodes*, as long as you have a black box function that can tell us whether the given node is a goal node. Suppose we generalize this problem to allow more than one possible start node. If your task is to find the shortest path between any start node and any goal node, can you solve this problem faster than merely running the BFS algorithm separately from each possible start node, and taking the minimum of the results of these individual searches?

Seven. Suppose each edge of some connected undirected graph is labelled with an integer. Your task is to find the shortest path from the given start node to any goal node (measured as the number of edges as in breadth first search, not by the sum of these edge labels as in Dijkstra's algorithm) under the constraint that the sum of labels on this path must be exactly divisible by *k*. How would you modify your graph searching algorithms to solve this variation? How does the running time compare to that of ordinary breadth-first search?

Lab 10: Just Plain Old Interesting Problems

One. Given a string that consists of only digit characters 0 to 9, and a positive integer `goal`, determine whether it is possible to break up this string into individual numbers whose sum is equal to `goal`. For example, the string "24519907323" could be broken into pieces 245, 19, 90, 732 and 3 to make up the goal value 1089.

Two. Given two sorted arrays of same length *n*, how quickly can you find the median value of their values combined together?

Three. Given an *n*-element array, find the longest contiguous subarray that does not contain any element twice. Can this be done in O(*n*) time?

Four. *Binary power exponentiation* is a nifty **repeated halving** algorithm and therefore quite efficient even for humongously large values of *n*. But contrary to the common misperception, the binary power algorithm is not optimal in that it would always reach the answer by making the smallest possible number of multiplications. For example, the binary power algorithm will perform a total of six multiplications to compute x^{15}, even though this quantity could be computed using only five multiplications when they are chosen more cleverly in a **multiplication chain** whose each step multiplies two terms that are known somewhere from the previous steps:

$x^2 = x^1 * x^1$
$x^3 = x^2 * x^1$
$x^6 = x^3 * x^3$
$x^{12} = x^6 * x^6$
$x^{15} = x^{12} * x^3$

Given *n*, use dynamic programming techniques to generate the shortest possible sequence of such multiplications to compute x^n. For each individual power x^k, you need to once again decide whether you are going to "take it" or "leave it" on your way to x^n, and again overlapping subproblems abound. (Note, for example, the futility of computing x^4 on the way to x^{15}, yet the binary power exponentiation algorithm computes this useless quantity anyway. But were you on your way to get to x^{16} instead, computing x^4 would become highly beneficial!)

This is also problem 122 in Project Euler. Wikipedia has a page "Addition chain exponentiation".

Five. We want to look for the given rectangular subarray inside a large two-dimensional array. How could Rabin-Karp algorithm be generalized to work more efficiently than brute force searching in the two-dimensional case?

Six. You are given a list of *axis-aligned rectangles* on the two-dimensional plane, each rectangle defined by its spans in the horizontal and vertical directions. Design an efficient algorithm to find a maximum subset of these rectangles so that there exists at least one common point (x, y) that is contained inside all rectangles of your chosen subset. What is the worst case asymptotic running time of your algorithm?

Seven. Recall that an *inversion* is a pair of indices i and j to an array A so that $i < j$ and $A[i] > A[j]$, that is, two elements that are out of order with respect to each other. The number of inversions, a rough measure how far away the entire array and its elements are from the sorted order, can be trivially computed in $O(n^2)$ time in the finest "Shlemiel" fashion with two nested loops to iterate through all pairs of positions in the array. However, how could the humble merge sort algorithm be modified to compute the number of inversions in $O(n \log n)$ time?

Eight. The insertion order of keys into an initially empty BST determines the structure of the tree. Suppose that each key x_i comes with the probability p_i that it tends to be searched for, and your task is to minimize the *expected search time for the key for a successful search*, the sum of terms $p_i d_i$ where d_i is the depth of the key x_i in the tree. Design an efficient dynamic programming algorithm to produce the optimum binary search tree structure for the given sorted set of keys $x_1, ..., x_n$ and their probabilities $p_1, ..., p_n$. (You need to choose which element x_i first becomes the root node, and then recursively build the optimal left and right subtrees from the elements $x_1, ..., x_{i-1}$ and $x_{i+1}, ..., x_n$. Note the repeated subproblems.) For simplicity, in this problem we only optimize the running time of the successful search, not the unsuccessful searches.

Nine. Sort the array elements in *descending frequency order* so that for two elements of the same frequency, the smaller element comes first. For example, given `[4, 6, 2, 2, 6, 4, 4, 4]`, the function would return `[4, 4, 4, 4, 2, 2, 6, 6]`. Write a solution that runs in linear time but uses linear extra space, and another solution that uses $O(\log n)$ space.