Lab 1: A Linear Mind

Zero. Search pattern is `"ABBAABBAA"`.

DFA transition table:

	0	1	2	3	4	5	6	7	8
A	1	1	1	4	5	1	1	8	9
B	0	2	3	0	2	6	7	0	2

Boyer-Moore good suffix function, from page "Boyer-Moore Demo":

A	B	B	A	A	B	B	A	A
12	11	10	9	12	11	10	2	2

Knuth-Morris-Pratt failure function, from page "Knuth-Morris-Pratt String Search":

0	1	2	3	4	5	6	7	8
A	B	B	A	A	B	B	A	A
0	0	0	1	1	2	3	4	5

One.

```
public static int maximumSubarray(int[] a, int k) {
    int sum = 0;
    for(int i = 0; i < k; i++) { sum += a[i]; }
    int maxSum = sum, maxLoc = 0;
    for(int i = 1; i < a.length - k + 1; i++) {
        sum = sum - a[i-1] + a[i+k-1];
        if(sum > maxSum) {
            maxSum = sum; maxLoc = i;
        }
    }
    return maxLoc;
}
```

Two.

```
public static boolean isJollyJumper(int[] a) {
    int n = a.length - 1;
    boolean[] jolly = new boolean[a.length];
    for(int i = 1; i < a.length; i++) {
        int diff = a[i] - a[i-1];
        if(diff < 1 || diff > n - 1 || jolly[diff]) { return false; }
        jolly[diff] = true;
    }
    return true;
}
```

Three.

```
public static int longestWithGivenSum(int[] a, int k) {
    int i = 0, j = 0;
    int maxLen = -1;
    int sum = a[i]; // Sum of subarray a[i..j]
    while(j < a.length) {
        if(sum >= k) {
            if(sum == k && j - i + 1 > maxLen) { maxLen = j - i + 1; }
            sum -= a[i++];
            if(i > j) { j = i; }
        }
        else {
            if(++j < a.length) {  sum += a[j]; }
        }
    }
    return maxLen;
}
```

Four.

```
public static boolean containsStrictMajority(int[] a) {
    if(a.length == 0) { return false; }
    int c = a[0]; // candidate for majority element
    int count = 1;
    for(int i = 1; i < a.length; i++) {
        if(a[i] == c) { count++; }
        else {
            if(--count < 0) { c = a[i]; count = 1; }
        }
    }
    // If strict majority element exists, c must be it now.
    count = 0;
    for(int e: a) { if(e == c) { count++; } }
    return 2 * count > a.length;
}
```

Five. To find the pattern inside the text whose suffix array has been constructed, use binary search to limit the left and right edges of the subarray of the suffix array whose suffixes start with the pattern.

To speed up the comparison of individual strings, the binary search can remember how many first characters are already known to match at both (left, right) edges of the binary search, and then compare characters only after that position. See the section "Applications" on the Wikipedia page "Suffix array".

Lab 2: Heaping Servings of Tasty Goodness

One. The alphabetical order can be always be trivially constructed from the frequency order, but not the other way around. In an array, the element can be accessed in O(1) when the position is given, but finding the position will take O(n) time when the element value is given. Returning the position instead of the element value will thus give the caller more power. Returning only a truth value about the existence of that element does not really empower the caller to do anything.

Two. First, a simple algorithm that will perform $1 + 2(n - 2) = 2n - 1$ comparisons in the worst case, realized when the array is already sorted in ascending order. Good enough for government work.

```
    public static int secondLargest(int[] a) {
        int m, mm; // largest and second largest so far
        if(a[0] < a[1]) { m = a[1]; mm = a[0]; }
        else { m = a[0]; mm = a[1]; }
        for(int i = 2; i < a.length; i++) {
            if(a[i] > mm) {
                if(a[i] > m) { mm = m; m = a[i]; }
                else { mm = a[i]; }
            }
        }
        return mm;
    }
```

But this famous problem can actually be solved using exactly n - 1 + ceil(log n) comparisons by first finding the largest element with the "cup tournament" arranged as a complete binary tree with n - 1 comparisons that are both necessary and sufficient for finding the maximum (why?), and then finding the maximum of only those elements that this maximum element beat along the way to victory. See this Stack Overflow page.

To quickly find which other elements were beaten by the winner during the cup, initialize an auxiliary array B[1..n] with values 0. Whenever the element in position i beats the element in the position j, assign B[j] := B[i] and B[i] := j. The chain of elements that were beaten by the winner k can then be iterated through in O(log n) time by following the indices starting from B[k] down the tournament cup tree.

Three. Use any interactive visualization of binary heaps, such as "Min Heap". (By the way, notice how cleverly this solution uses the otherwise unused position zero of the array as a **sentinel** so that the siftup method does not need to check that you are still inside the array. And here I was using that position to store the number of elements, performing one extra comparison in every round of the loop like some chump. Live and learn!)

Four. Copy both arrays into new array of size $n + m$, and run `Build-Heap` for the new array. Running time is O($n + m$). The part of A and B already being heaps was there only as a lure to entice you to think that you are somehow required to exploit their already heap-ness in this problem, but the correct solution is to just ignore that order altogether and build the result heap essentially from scratch.

Five. Ditto. Just run `Build-Heap` using the min-heap discipline instead. Running time is the same O(n) as before. Note that this problem cannot be solved by simply reversing the array.

Six. `Left(i) = d * i`, `Parent(i) = floor(i / d)`. When pushing a new element into the d-ary heap, the element rising up to its final resting place needs to be compared only to its parent at each level, and the number of siblings is irrelevant. `Push` is therefore O($\log_d n$), which is asymptotically equal to O(log n) since all logarithms are a constant factor away from each other, but `Extract-Max` is now O($d \log_d n$) because when an element is sinking down, we need to look at all d children of the current position to decide where to continue.

Seven. Ditto, again. Copy all elements into a whole new array and run `Build-Heap` in total asymptotic running time O(nm).

However, this problem suddenly becomes much more interesting if you are supposed to find only the overall smallest k elements of all these min-heaps. Maintain a secondary "heap of heaps" whose elements are heap identifiers, sorted using a *custom comparator* that checks which heap currently has a smaller element at its root. This "heap of heaps" can be used to find the heap whose head element is currently the smallest of all elements that still remain in the heaps. Extract that element from that heap and append it to the result, repair the contents of that one heap the usual way, and push the identifier of that heap back into your "heap of heaps". The total running time of this scheme is only O(n + k (log nm)), with the first term O(n) coming from building the initial heap of heaps. Clever!

Lab 3: Linked Data Structures

One. See this Stack Overflow page for solutions.

Two. Our first instinct might be to maintain a hash table or some other dynamic set of the nodes to remember which nodes we have already seen, and each time when moving forward in the list, check if the current node is already in that dynamic set. That would work, but this problem has a way more interesting solution that needs only O(1) extra memory!

Use two pointers "tortoise" and "hare", both starting off at the first node. Each round, the tortoise moves one step forward, whereas the hare moves two steps. If the hare reaches the end of the chain by following a `next` reference that is `null`, the chain is a proper singly linked chain and we can stop without having to wait for the tortoise to also reach the end. Otherwise, if the chain turns into a self-loop at some point so that the tortoise and hare would be doomed to run around in a circle forever, eventually also the tortoise will follow the hare into that loop, after which the hare must necessarily eventually catch up with the tortoise from behind. (Think about it.) If that happens, the algorithm can stop and report that the chain contains a self-loop. Either way, the algorithm will terminate after after O(n) steps.

(This same idea can also be used to split a singly linked chain into two halves of equal size, plus minus one, in a single pass. As an aside, how would you modify this algorithm to report not just the existence but the actual length of the loop?)

Three. Maintain an array of head pointers to k + 1 separate chains of elements arranged in a cyclic fashion 0, ..., k. These chains contain the elements whose value is that much higher than that of the current minimum element in the priority queue. `Push` is O(1) by adding a new element in the correct list, `Pop` is O(1) from popping from the list whose cyclic index is currently 0. If popping the element makes that chain empty, but the queue itself is nonempty, the cyclic pointer is moved forward until a nonempty chain is found.

Four. Another classic chestnut. Implement `Push` to simply push the element in the first stack. The `Pop` operation pops from the second stack, if there is at least one element there. Before that, if the second stack is empty, move all the elements from the first stack to second using `Pop` and `Push`. This makes the queue `Pop` operation to be O(n) in the worst case, since n elements may have to be moved from the first stack to the second just to pop out the next one coming out of the queue, but this can happen only if n elements have been first added to the first stack using O(1) operations, over which this O(n) copying can then be amortized.

Five. Store the nodes that contain the elements of the current stack in a singly linked chain, with the pointer to the first node representing the stack. The operation `Push` creates a new node to the front of the current chain and returns that node. The operation `Pop` returns a pointer to the second node of the chain, **but does not remove the first node**. After any series of `Push` and `Pop` operations, the resulting structure becomes a branching *parent tree* whose first nodes share the same chains for the rest of the stack. This way each new version of the stack needs only O(1) extra memory, not O(n). (This is also, by the way, how lists are internally represented in *Prolog logic programming language* that you will meet in CPS 721, to keep the list data structure immutable but allow O(1) logical operations in the front.)

This same idea can later be used for arbitrary tree data structures so that the new and the old version of the tree share all their existing branches that did not change in the operation that created the new version, so the new version has to create new tree nodes only for those branches that are somehow different from how they used to be in the old version. This allows efficient data structures in *pure functional programming languages* that do not allow any destructive assignment, the defining hallmark of imperative programming. See the Wikipedia article "Purely functional data structures" for more information.

Six. Maintain an array of 365 head pointers, one pointer for each day of the year. Each list contains the future events for that day, sorted by their ascending year. To `Push` an event, insert it to the list of the day of that event, iterating through the list until you find the correct position where the event falls for its year. Outside this structure, maintain the current day and current year. The operation `Pop` steps cyclically forward in the list from the current day until the first element of that chain takes place in the current year.

Note that this data structure has a rather curious property that its chains can be arbitrarily long but the nodes hanging at after the first element of the chain consume zero time in the `Pop` operation, unlike they would in binary heap and other similar priority queue structures that need reorganizing after each `Pop`. Even if the rest of our entire physical universe were somehow hanging from these chains, `Pop` would still take as much time as is needed to step into the next event.

Lab 4: Comparison sorting

One. Various possible solutions, in order of increasing asymptotic efficiency:

1. Sort the entire array, then just list the top k elements: O($n \log n$)
2. Turn array into binary heap, call `Extract-Max` k times: O($n + k \log n$)
3. Find the k:th largest element, `Partition`, sort the top k elements: O($n + k \log k$)

Two. Turn each key *x* into a pair (*x*, *i*) where *i* is the original position. In case that keys are equal, resolve their comparison by comparing these original positions. You need O(*n*) extra space. For the second part, make the algorithm first run through the array to check if it is already sorted, and do nothing if this is the case.

Three. See the explanation for item 2-1, "Disturbed Array", on the page "Algo Muse".

Four. Implement the operation `split` that reads in the given file one line at the time, writing these lines in two separate new files in alternating fashion. This program needs to keep in memory only one line at any given time. Then, implement operation `merge` that reads in two files one line at the time, writing the line that is earlier in the sorted order into the result file, advancing in that file and staying put in the other. This program needs to keep in memory only two lines at any given time. Implement the recursive merge sort using these operations as subroutines.

Five. If some comparison-based priority queue allowed O(1) operations `Push` and `Extract-Max`, it would allow comparison-based sorting in O(*n*) time, which is mathematically impossible. At least one of these two operations must necessarily be logarithmic in the worst case.

Six.

n	Sorting network (square brackets denote a group parallelizable operations)
2	`Cex(1, 2)`
3	`Cex(1, 2), Cex(1, 3), Cex(2, 3)`
4	`[Cex(1, 2), Cex(3, 4)], [Cex(1, 3), Cex(2, 4)], Cex(2, 3)`

If you add `Cex(2,3)` between the first two square bracket groups of network for *n* = 4, the resulting network will no longer sort all inputs correctly. For example, it will fail for the input `[4,1,2,3]` whose largest element 4 will finish up in second last position, instead of the correct last position.

Seven. This classic "nutcracker" puzzle of algorithmic thinking can be solved with a simple variation of quicksort. Choose a random nut and use it to partition the bolts into those that are too small for it, the one that matches the nut exactly, and those that are too large for it. Then use that exactly matching bolt discovered in the previous operation to similarly partition all the nuts. Appreciate for a moment how the resulting partitions of nuts and bolts will now be the exact same sizes both ways, so you can recursively match all the smaller nuts and bolts with each other, and similarly match the all larger nuts and bolts.

The same techniques that make quicksort run in the guaranteed O(*n* log *n*) worst case time without any randomization will also make this algorithm run in the guaranteed O(*n* log *n*) time.

Lab 5: Unconventional sorting

One. LSD Radix sort:

	After column 1	After column 2	After column 3
BOB	BOB	TAB	BAT
TAP	TAB	TAP	BOB
TOP	TOP	BAT	CAT
BAT	TAP	CAT	COT
COT	BAT	BOB	TAB
TAB	COT	TOP	TAP
CAT	CAT	COT	TOP

MSD Radix sort:

	Split on column 3	Split on column 2	Split on column 1
BOB	BOB	BAT	
TAP	BAT	BOB	
TOP	COT	CAT	
BAT	CAT	COT	
COT	TAP	TAP	TAB
TAB	TOP	TAB	TAP
CAT	TAB	TOP	

Two. Each element *x* is inserted into the bucket `floor(F(x) * n)`, where the function *F* is the CDF of the elements. Note that when the elements are drawn randomly from uniform distribution, we have `F(x) = (x-a) / (b-a)` and the previous formula produces the bucket index of the ordinary bucket sort as this special case of all possible distributions.

Three. Discuss this one with students and try to activate their thinking caps and get some good discussion and analysis going. Note that the thin metal rods can be sorted in nearly O(1) time by grabbing them all in your fist and pressing them against the surface of a flat table so that their bottom ends are all at the same level.

Four. Here is an example implementation in Java:

```
private static int median3(int[] a, int i) {
    if(i == a.length - 1) { return a[i]; }
    if(i == a.length - 2) { return a[i]; }
    if(a[i] < a[i+1] && a[i] < a[i+2]) {
        return a[i+1] < a[i+2] ? a[i+1] : a[i+2];
    }
    else if(a[i] > a[i+1] && a[i] > a[i+2]) {
        return a[i+1] > a[i+2] ? a[i+1] : a[i+2];
    }
    else { return a[i]; }
}

public static int tukeysNinthers(int[] a) {
    if(a.length < 4) { return median3(a, 0); }
    int[] medians = new int[(a.length + 2) / 3];
    for(int i = 0; i < a.length; i += 3) {
        medians[i/3] = median3(a, i);
    }
    return tukeysNinthers(medians);
}
```

The above algorithm could be made to work in O(1) extra space by moving each median-of-three to the initial prefix of the array, with the help of an index variable *next* that shows where the next median-of-three is to be moved.

Here is a randomized sampling algorithm that estimates how often each element gets chosen to be the median:

```
public static void estimateTukeys(int n, int rounds) {
    Random rng = new Random();
    int[] results = new int[n];
    int[] a = new int[n];
    for(int i = 0; i < n; i++) { a[i] = i; }
    for(int i = 1; i < rounds; i++) {
        for(int j = 0; j < n; j++) { // Knuth shuffle
            int k = rng.nextInt(j + 1);
```

```
                int tmp = a[k];
                a[k] = a[j];
                a[j] = tmp;
            }
            results[tukeysNinthers(a)]++;
        }
        for(int i = 0; i < n; i++) {
            if(results[i] > 0) {
              System.out.println(i + " selected " + results[i] + " times.");
            }
        }
    }
```

An example run with `n = 21` and `rounds = 100000` shows that for the vast bulk of the 21! possible array permutations, the estimate returned by Tukey's ninthers algorithm is close to true median value 10, which was returned in one fourth of all permutations. Note also how no input permutation of these 21 numbers, even if Old Nick himself were your adversary in this problem to set them up in the worst possible fashion, can ever fool Tukey's ninthers algorithm to select an estimate for the median that is less than five or higher than fifteen.

```
5 selected 298 times.
6 selected 1457 times.
7 selected 4614 times.
8 selected 11016 times.
9 selected 19931 times.
10 selected 25070 times.
11 selected 20041 times.
12 selected 11205 times.
13 selected 4565 times.
14 selected 1497 times.
15 selected 305 times.
```

Five. You can just use counting sort, which is stable, linear, and uses O(1) memory. However, if stability is not needed, the Hoare partitioning algorithm can be modified to collect all zeros to the beginning and all ones to the end. Skip over all zeros from left, and skip over all ones from right, swapping the ones and zeros that you encounter, and continue for the remaining middle.

Six. In both quicksort and mergesort, the recursive calls for disjoint subarrays are completely independent of each other, so one of them can be passed to another core while the current core continues to solve the other. However, the merge step cannot begin until both subarrays have been completely sorted. (Of course, any decent parallelism frameworks will have the processor core that finished first already sorting some other subarray instead of twiddling its thumbs and going "la la laaa".)

Quicksort does not have to wait for the other subarray, but there has to be one general wait for all subarray sorting tasks to have completed until the top level method call can return and declare the entire array can be sorted. MSD radix sort can be parallelized analogous to quicksort.

LSD radix sort, heapsort, selection sort and insertion sort do not parallelize easily, since these parallel sorting tasks would be trampling all over each other while they race to modify the contents of partially or fully overlapping subarrays. The locking required to prevent these race conditions would probably eat away all the gains of parallelism.

Lab 6: Hashing

One. This scheme turns out not to be that big of a help. The length of chain is denoted by *k*.

Operation	Unsorted	Sorted
`add`	O(1)	O(*k*)
`remove`	O(*k*)	O(*k*)
`contains`	O(*k*)	O(*k*)

Two. The idea is to modify the basic rejection sampling algorithm to treat every chain as if it were of length *L*. Choose a random slot and random integer *k* from 0, ..., *L* - 1. If the *k*:th element exists in the chain, return that, otherwise start again. As in rejection sampling in general, the running time can be potentially unlimited, but obeys exponential distribution so the probability that it gets too high is negligible.

Three. Each node should have pointers `next` and `prev` to organize these nodes in doubly linked list, separately from the chains hanging from the slots of the hash table. Modify the insert and remove operations of the entire hash table to update these pointers.

Four. To encode the board that has 64 squares that may each contain a black or a white piece, first use 64 bits to encode the **propositions** "Square s_i contains a black piece" and then another 64 bits to encode the propositions "Square s_i contains a white piece". So *k* = 128 is enough (even a "bit" redundant, if each square can contain at most one kind of piece) to encode any possible board position.

Quickly updating the hash value for the board position is based on the really useful property of exclusive or that `(a xor b) xor b == a` for any two integers `a` and `b`. This allows us to quickly compute the hash function for the position that results from one move. For the example at hand, take the hash value h_1 for the current position, and compute its exclusive or with the elements of array A in the positions that correspond to the proposition "Square s_1 contains a black piece" and "Square s_2 contains a white piece" to effectively remove these pieces. Compute the exclusive or of the result with the element of A in the position that corresponds to the proposition "Square s_2 contains a black piece". This way, the hash code h_2 for the new position can be computed in O(1) time with only three bitwise arithmetic operations.

Five. Keep searching down the tables, looking at only the one slot at each overflow table that can contain the element that you are searching for, until you either find the element that you are looking for, or hit an empty slot or deleted slot. If the current slot that you are looking at has some other key, continue down to the next lower overflow table. The running time of all operations is O(k), except that in theory, adding a new element can trigger expansions until the size of the entire table is 2^n, big enough to guarantee that there is a slot available for every element. The probability of this or anything even remotely close to that happening is utterly negligible.

Lab 7: Dynamic Programming

One. In mergesort and quicksort, each recursive call decreases the subproblem size to half of the size of the previous level, so there are at most O(log n) levels. Adding up the total work at these levels gives us O(n) * O(log n) = O(n log n), either by using the Master Theorem or, in the case of this particular recursion, a simple geometric argument for the area of a rectangle. In Towers of Hanoi and similar recursions, each recursive call decreases the subproblem size by only 1, producing an exponential time recursion.

Unlike the problems with exponential number of paths through polynomial number of states that can be tamed with dynamic programming, the state space of Towers of Hanoi is exponentially large, and its subproblems do not repeat in different branches of the recursion.

Two. Let K(s) be the key in node s. Actually, an even better recursion to solve is M(s, t) where t is boolean valued parameter telling whether the parent of s was already taken in.

M(s, true) = Sum(M(s', false))
M(s, false) = max(K(s) + Sum(M(s', true)) , Sum(M(s', false))

(In all Sum formulas, the variable s' iterates through all children of s.)

With the same idea, we can also make up and solve variations of this problem such as "At most one child of any given node can be taken in".

Three. Let I(n, m) be the boolean function telling if the prefix of C of length (n+m) can be constructed by interleaving the prefixes of A and B of length n and m, respectively.

I(n, 0) = true iff A[1..n] = C[1..n]
I(0, m) = true iff B[1..m] = C[1..m]
I(n, m) = (A[n] = C[n+m] and I(n-1, m)) or (B[m] = C[n+m] and I(n, m-1))

These induce the following dependency chart (green = base case, blue = needed to fill in the value of the cell denoted by ???) that is trivial to fill with two nested for-loops.

n \ m	0	1	2	3	4	5
0	🟩	🟩	🟩	🟩	🟩	🟩
1	🟩					
2	🟩					
3	🟩					
4	🟩					🟦
5	🟩				🟦	???

Four. Let S(n, g) denote that it is possible to make up goal g from the first n elements of the array A. The recursive equations are:

S(n, 0) = true
S(0, g) = false
S(n, g) = S(n-1, g-A[n]) or S(n-1, g)

This induces essentially the same dependency chart as the problem of 0-1 knapsack.

Five. The recursive equations for W(n) are:

W(n) = 1 if n < 3
W(n) = W(n - 1) + Sum $_{2 \le k \le n}$ W(n - (k + 1))

The first term of the second line is the number of ways to fill starting with a black tile, and the second term is the number of ways to fill starting with a red tile whose length has to be between 3 and n, inclusive, followed by a black tile. (The last black tile is imaginary if k = n.) The dependency chart would have an arrow from the current square (denoted by ???) to all previous squares down to 2.

n	0	1	2	3	4	5	6
	🟩 1	🟩 1	🟩 1				???

Lab 8: Binary Search Trees

One. Use the interactive visualization on the page "Binary search trees".

Two. If you simply insert the sorted elements into an empty tree one by one, you will end up with a maximally unbalanced tree. No good. A better solution is to convert the subarray to a binary search tree recursively by making the middle element to be the root node (we might as well, to allow the resulting structure to be maximally balanced), and then recursively convert the left and right subarray to trees and make these trees the left and right child of this root. Maybe something like this (where objects of type `BST` are the individual nodes):

```
    public static BST createTree(int[] a, int start, int end) {
        if(start > end) { return null; }
        int mid = (start + end) / 2;
        BST root = new BST(a[mid]);
        root.left = createTree(a, start, mid-1);
        root.right = createTree(a, mid+1, end);
        return root;
    }
```

Note that the same way as in the recursive tree walk algorithm, the running time of `createTree` is only O(n) instead of O(n log n), since the same constant amount of work is performed for each node regardless of its position in the structure. Inserting the new elements one by one into an initially empty tree structure would produce an O(n log n) algorithm.

Three.

```
public static void rangeSearch(
    BST root, // node that we are currently in
    List<Integer> result, // accumulated result
    int min, // lower bound of range
    int max  // upper bound of range
) {
    if(root == null) { return; }
    if(root.key > min) {
        rangeSearch(root.left, result, min, max);
    }
    if(min <= root.key && root.key <= max) { result.add(root.key); }
    if(root.key < max) {
        rangeSearch(root.right, result, min, max);
    }
}
```

Four. Scroll down the page "Enumerating trees".

Five. Modify the recursive tree walk algorithm to take two additional parameters *min* and *max* that constrain the possible keys that may exist in the current branch, as determined by the keys in the path of ancestor nodes leading into the current node. At the top level call at the root, there are no constraints so *min* = -∞ and *max* = +∞. When arriving into a node, check that its key *k* lies between *min* and *max*, exclusive, and stop with the negative answer otherwise. Thereafter, in every recursive call that descends to the left child, update *max* to equal *k* and keep *min* at its current value. Symmetrically, when descending to the right child, update *min* to equal *k* and keep *max* at its current value.

As an aside, many of these kinds of algorithms expressed in high level pseudocode use -∞ and +∞ as sentinel values that are guaranteed smaller and larger than any actual value that exist in the data. But of course we know, for example, that the Java `int` type and similar do not have such values. (Floating point types conveniently do allow both infinities.) When implementing the previous algorithm in Java to operate on a BST whose nodes contain `int` keys, how would you represent these initial boundary values with the tools that actually exist in the language?

Six. Base case of induction is true, since AVL trees of height zero and one contain at least one node. Assume then that the induction hypothesis holds for all integers up to *h*, and consider any AVL tree of height *h* + 1. One of the subtrees of the root node must have the height of exactly *h*, and thus contain at least F_h nodes by the induction hypothesis. The other subtree must have the height of at least *h* - 1, and thus also contain at least F_{h-1} nodes by the induction hypothesis. Therefore the entire AVL tree must have at least $1 + F_h + F_{h-1} = 1 + F_{h+1}$ nodes.

Lab 9: Graphs

One. We are looking at the fifteen-puzzle, the dusty old cliche for teaching A*. But once again, like all cliches, this one also became a cliche for a good reason. Behold:

(a) The state space of the 15-puzzle consists of the 16! possible ways to place the tiles and the empty tile into the 4-by-4 square. However, as explained on the Wikipedia page, this state space graph consists of two connected components that are symmetric but unreachable from each other, as proven by a clever **invariant argument**, a clever technique to prove properties of large state machines (including computer programs).
(b) Each state has two, three or four possible moves emanating from it, depending on whether the empty tile is in a corner square, edge square or inside square.

(c) Several simple heuristics have been devised for this classic introductory problem of graph search. The most useful technique to do this for you to learn is **constraint relaxation**, because it can be applied to any other problem where some set of constraints determines whether a move is legal.

Removing some of these constraints cannot possibly make the puzzle harder, since any legal solution to the original puzzle is also a legal solution to the relaxed puzzle. Therefore the solution of the relaxed puzzle cannot be longer than the solution of the original puzzle, and we can use the solution of the relaxed puzzle as the heuristic function. We need to drop enough constraints so that the heuristic function can be evaluated on the spot for the given state s.

In the 15-puzzle, we can simply drop the constraint that no two tiles can simultaneously lie on the same square. The solution to the heuristic problem is now the sum of Manhattan distance of each tile to its goal position, and this sum can therefore be used as the heuristic function for the original, more constrained problem.

As a nice little bonus (sort of similar to Zobrist hashing in spirit), when the A* algorithm expands the state s_1 to find the neighbouring state s_2, the heuristic for the new state s_2 can be computed easily from the known heuristic for the state s_1, instead of computing it from scratch by looping through all tiles and adding up their Manhattan distances.

Two. First, compute and store the in-degree of each node. Initialize a queue with the nodes whose in-degree is zero. Repeatedly, pop one node from the queue, give it the next running number, and decrement the in-degree of its every neighbour by one. If the neighbour's in-degree becomes zero, add it to the queue. This numbers all the nodes in $O(V + E)$ time without actually modifying the underlying graph.

If the graph has a cycle, the in-degrees of the nodes of this cycle, and all the nodes reachable from this cycle, will never become zero. The algorithm will number the nodes that it can, and will then stop the moment that the queue becomes empty, with the nodes in the cycle and all the nodes reachable from them left without a number. This is basically what happens when *garbage collection* of dynamically allocated objects relies only of *reference counts* of how many other objects point to the particular object, and releasing the object when the count becomes zero. A cycle of zombie objects can keep each other alive, and worse, keep alive all the objects outside the cycle that they point to.

Three. Without loss of generality, assume the edge e is in T_1 but not in T_2. Adding the edge e to T_2 would necessarily create some cycle T_2 that e is part of. There must exist at least one edge e' in T_2 that is in this cycle but is not in T_1. (Otherwise, T_1 would already contain that same cycle of edges, which is impossible since T_1 is a spanning tree.) Since edge lengths are unique and e was intentionally chosen to be the shortest edge that is not included in both MST's, the length of e' is longer than the length of e. But now removing e' from the T_2 and replacing it with e would create a new MST that is better than T_2, a contradiction.

If the edge lengths are not unique, e and e' can have the same length and work equally well in creating an MST.

Four. The MST of the graph does not change. The length of any spanning tree will increase by the exact same total amount $c(V - 1)$, and therefore such a change does affect which spanning trees happen to be minimal.

Corollary 1: No, since you can always add the same big enough positive constant C to all edge lengths to make them all to be positive without affecting the MST.

Corollary 2: To find the maximum spanning tree of the given graph, negate the edge lengths and compute the MST, then negate the edge lengths back. (More generally, in all problem domains that allow the negation operator, minimization and maximization are the exact same problem simply by negating the signs of the quantities involved! The minimum of $f(x)$ is always equal to the maximum of $-f(x)$, and vice versa. However, should a problem inherently contain only positive integers, as many NP-complete graph optimization problems do, this technique will no longer apply, and the minimization and maximization versions of the same problem can be rather different in their inherent difficulty!)

Five. (a) Finding some cycle in a connected undirected graph can be done with either DFS or BFS in time $O(V)$, assuming the edge-list representation. (After looking at most V edges, no matter which nodes they emanated from, some node must have repeated, thus revealing a cycle.) This cycle contains at most V nodes. Therefore, the total running time is $O(V^2)$. (b) Sorting the edges takes $O(E \log E)$ time. For each edge, checking that graph remains connected after removing it can be done in $O(V)$ time, giving us a straightforward running time $O(E \log E + VE)$. As the Wikipedia page points out, the second term can be made much lower by using the union-find data structure to test for connectivity.)

Six. All you have to do is add one new "superstart" node to the graph, and add a directed edge of zero length from the superstart to all you actual start nodes. Then run the ordinary BFS or Dijkstra once from the superstart node. (This same simple idea works just as well for many other graph structural optimization problems such as *maximum flow*.)

Seven. Perform BFS in a modified graph whose nodes are all pairs (n, m) where n is some node in an original graph and m is an integer in $\{0, ..., k - 1\}$. In this graph, there is a transition from (n_1, m_1) to (n_2, m_2) if and only if the original graph contains the edge (n_1, n_2) whose cost c satisfies the equation $m_2 = (m_1 + c) \bmod k$. The start node is $(s, 0)$, and the goal nodes are $(n, 0)$ where n is a goal node of the original graph. The running time of this algorithm is $O(k(V + E))$.

Lab 10: Just Plain Old Interesting Problems

Eight. Let P(i, j) denote the sum of probabilities of keys x_i, ..., x_j. The average search time for a key in a BST constructed from keys x_i, ..., x_j is given by the recursive equation

$S(i, i) = 1$
$S(i, j) = \max_{i \le k \le j} (p_k + P(i, k\text{-}1)(1 + S(i, k\text{-}1)) + P(k\text{+}1, j)(1 + S(k\text{+}1, j))$

In the second formula, the three terms correspond to the cases "the element is in the root" (which happens with probability p_k), "the element is in the left subtree", and "the element is in the right subtree". The dependency graph induced by these equations is familiar looking:

i \ *j*	1	2	3	4	5	6
1	1	🟦	🟦	🟦	🟦	???
2		1				🟦
3			1			🟦
4				1		🟦
5					1	🟦
6						1