Exercises 3

Good problems

KT 4.5

Our algorithm works as follows. Let the houses be at locations $x_1 \dots x_n$, We place the first base station at $x_1 + 4$. We then iterate through each house in ascending order. At house x_i, if x_i is not in range of the last base station, we place a base station at house $x_i + 4$.

```
function stations(x_1 ... x_n) {
    lastStation = x_1 + 4
    stations = [ lastStation ]
    for i = 2 to n
        if (abs(x_1 - lastStation) > 4)
            lastStation = x_1 + 4
            stations.push(lastStation)
    return stations
}
```

Correctness: Suppose there is a different optimal algorithm that returns a solution y = $[y_1, \dots y_k]$. Let s = $[s_1, \dots s_j]$ be the solution returned by our algorithm. Let c be the first index where y_c and s_c differ. Since our algorithm always places stations 4 miles after the next unserviced house and both solutions are the same before this station, neither solution has a station in range of the house at c. Hence the tower at y_c and s_c must be in range of the house. Since s_c = house + 4, we must have that $y_c < s_c$, otherwise it would not cover this house and y would not be optimal. We can then exchange y_c with s_c and y would still be optimal, as it would have all houses covered with the same number of stations. We can repeat this with all differences and since after every exchange y is still optimal, we must have y = s.

KT 4.7

We process the jobs in non-increasing order of finish time.

Runtime: It takes O(nlogn) time to sort the jobs.

Correctness: Let O be an optimal schedule where there is an inversion (a pair of jobs J_i and J_j where J_i is scheduled immediately before J_j and $f_i < f_j$). Let O' be the same schedule as O except with J_i and J_j swapped. We show that O' has a completion time smaller than or equal to O. The combined time spent on the supercomputer by J_i and J_j in preprocessing in O is $p_i + p_j$, which is the same as that of O'. Hence swapping the order does not change the completion time of any other jobs. In O, J_j completes later than J_i, since it is preprocessed later and has a greater finish time f_j. Let t be the completion time of J_j in O. In O', J_j is preprocessed first, so has a completion time of less than t. J_i completes preprocessing in O' at the time in which J_j finished preprocessing in O, but since it has a smaller finish time f_i, has a completion time of less than t. Hence the completion time of O' must be smaller than or equal to O, and performing multiple swaps in this way will lead us to the same schedule as our algorithm. Hence our algorithm is optimal.

KT 4.15

Our algorithm works as follows. We initialize a S to be the set of all intervals. We sort S in non-decreasing order of end time. We iterate through each of the intervals in S. For each interval j,

we find all intervals in S that intersect j. We take the one with the latest finish time, k, and add it to our result. Then we find all intervals in S that intersect k and remove them from S.

Runtime: Sorting the intervals stakes O(nlogn) time. Iterating through each interval takes O(n) time. In each iteration, it takes O(n) time to find the intervals that intersect j, and O(n) time to find the intervals that intersect k. Hence our runtime is O(n^2).

Correctness: Let D = $d_1 \dots d_l$ be an optimal committee, sorted by shift end time. Let C = $c_1 \dots c_m$ be the committee returned by our algorithm, sorted by shift end time. Let k be the first index where c_k is not in D. We show that we can swap c_k with some element in D so that $c_1 \dots c_k$ is in D, and it is still a valid committee. Let x_k be the interval of the iteration that added c_k to C. D must contain an interval d_j that covers x_k. d_j is not in $c_1 \dots c_{k-1}$, since x_k was uncovered in the iteration that added c_k. We create D' by removing d_j from D and adding c_k. By construction, we have that D' is the same size as D and contains $c_1 \dots c_k$. Assume for sake of contradiction there is some interval x not covered by D'. Then it must not be covered by $c_1 \dots c_k$, since those are all in D'. This means it finishes later than x_k. Then we have $finish(d_j) \geq finish(c_k) < start(x)$. Hence x is not covered by d_j. But this means x is not covered by any interval in D, which is a contradiction. Hence D' is a valid committee. We can perform swaps in this way until we reach the committee produces by our algorithm.

KT 4.17

This is the interval scheduling problem but circular. Let S be the set of intervals that run across midnight. For each interval j in S, we remove all intervals that overlap j, and run the normal interval scheduling algorithm on the remaining intervals. We take the maximal schedule of each of these schedules.

Runtime: The number of intervals running across midnight is in O(n). It takes O(n) time to remove intervals intersecting j. Interval scheduling takes O(nlogn) time. We can remove the need to sort in every iteration in interval scheduling by sorting the intervals at the beginning of the algorithm. Then our algorithm takes O(nlogn + n^2) which is in O(n^2) time.

Correctness: There can only be one job that runs at midnight in any valid schedule. We try all possible jobs that run at midnight and take the best schedule out of them. Hence our algorithm is correct.

DPV5.25

We can model this as a graph problem G, where every variable is a vertex. There is an edge between two vertices if there is an equality constraint between them. All variables in a connected component must all be equal. Hence our algorithm works as follows. We run DFS to obtain all the connected components of G. We assign a unique number to each connected component, and record for each variable which number it is assigned. Then we iterate through all the disequality constraints and check if the two variables are assigned different numbers. If this is true for all disequality constraints, return true. Otherwise it is not possible and we return false.

Runtime: G has n vertices and m edges. It takes O(m+n) time to construct the graph. DFS takes O(m+n) time. Iterating over each disequality constraint takes O(m) time. Hence our algorithm takes O(m+n) time.

Correctness: Two variables are in the same connected component if and only if they must be equal to satisfy the equality constraints.

DPV4.15

[variation of Midterm problem]
We use a modified version of Djikstra's algorithm to solve this problem. We initialize an array numpaths for each node u to all 0s, except numpaths[s] = 1.

```
function solution() {
    dist(v) = infinity for every v in V
    dist(s) = 0
    numpaths(u) = 0 for every v in V
    numpaths(s) = 1

    Q = PriorityQueue(V) // dist(v) is the key
    while Q is not empty
        u = Q.deleteMin()
        foreach neighbor v of u
            if dist(u) + l_uv < dist(v)
                dist(v) = dist(u) + l_uv
                Q.decreaseKey(v, dist(v))
                numpaths[v] = numpaths[u]
            else if dist(u) + l_uv == dist(v)
                numpaths[v]++

    usp(v) = true if numpaths(v) == 1, else false for every v in V
    return usp
}
```

Runtime: We only added O(1) operations to the loop (updating numpaths). Djikstra's algorithm takes O((|V| + |E|)log|V|) time. Hence our algorithm has the same runtime.

Correctness: We prove correctness of the algorithm by induction on dist[.]. If dist[v] = 0, then v = s and numpaths[s] = 1, since there is one path from s to itself. Let dist[t] = k, and assume for vertices with dist[.] < k that the algorithm computes numpaths[.] correctly. Every path of length k from s to t must pass from a neighbor y of t with distance k - l_{yt}. We set numpaths[t] to numpaths[y] for each y with a shorter path than found previously, since the number of paths to t is the same as the number of paths to y. Otherwise, if we find another shortest path, we increment the number of shortest paths. Hence numpaths[t] is correct.

DPV4.18

We use a modified version of Djikstra's algorithm to solve this problem.

```
function solution() {
    dist(v) = infinity for every v in V
    dist(s) = 0
    best(u) = infinity for every v in V
    best(s) = 0

    Q = PriorityQueue(V) // dist(v) is the key
    while Q is not empty
        u = Q.deleteMin()
        foreach neighbor v of u
            if dist(u) + l_uv < dist(v)
                dist(v) = dist(u) + l_uv
                Q.decreaseKey(v, dist(v))
```

```
                    best[v] = best[u] + 1
                else if dist(u) + l_uv == dist(v)
                    best[v] = min(best[v], best[u]+1)
        return best
}
```

Runtime: We only added O(1) operations to the loop (updating best). Djikstra's algorithm takes O((|V| + |E|)log|V|) time. Hence our algorithm has the same runtime.

Correctness: We prove correctness of the algorithm by induction on dist[.]. If dist[v] = 0, then v = s and best[s] = 0, since there are no edges on a shortest path from s to itself. Let dist[t] = k, and assume for vertices with dist[.] < k that the algorithm computes best[.] correctly. Every path of length k from s to t must pass from a neighbor y of t with distance k - l_{yt}. We set best[t] to best[y]+1 for each y with a shorter path than found previously, since the number of edges to t is the number of edges to y plus 1. Otherwise, if we find another shortest path, take the minimum number of edges of the 2 paths. Hence best[t] is correct.

DPV4.19

We use a modified version of Djikstra's algorithm to solve this problem, where we also add the vertex costs to the distance during computation.

```
function solution() {
    dist(v) = infinity for every v in V
    dist(s) = c_s

    Q = PriorityQueue(V) // dist(v) is the key
    while Q is not empty
        u = Q.deleteMin()
        foreach neighbor v of u
            if dist(u) + l_uv + c_v < dist(v)
                dist(v) = dist(u) + l_uv + c_v
                Q.decreaseKey(v, dist(v))
    return dist
}
```

Runtime: We only added O(1) operations to the loop (updating the calculation to include vertex cost). Djikstra's algorithm takes O((|V| + |E|)log|V|) time. Hence our algorithm has the same runtime.

Correctness: We prove correctness of the algorithm by induction on dist[.]. If dist[v] = c_s, then v = s and best[s] = 0, we just have the vertex cost of s on a path to itself. Let dist[t] = k, and assume for vertices with dist[.] < k that the algorithm computes dist[.] correctly. Every path of length k from s to t must pass from a neighbor y of t. dist[y] is correct by our inductive hypothesis. Then the cost of the path to t is just the cost of the path to y, plus the edge cost l_{yt} and the vertex cost of t. Hence our algorithm is correct.

DPV 5.5

A) No. Since all the edge weights are non-negative, incrementing every edge weight by one means every MST still has the same relative ordering of total weight to each other.

B) No. Since all the edge weights are non-negative, incrementing every edge weight by one means every path still has the same relative ordering of total weight to each other.

DPV 5.6

Suppose G has two distinct MSTs T and T'. Each tree must contain an edge the other does not. Let e be a minimum weight edge in T \ T', and e' be a minimum weight in T' \ T. Without loss of generality, suppose $w_e \leq w_{e'}$.

We can add e to T' to create a cycle C. Let e'' be any edge of this cycle that is not in T (we may or may not have e'' = e'). e'' is in T' \ T, since it is in T' but not T. Then we must have that $w_{e''} \geq w_{e'} \geq w_e$ since e' is the minimum weight edge in T' \ T.

Then we can create a spanning tree T'' = T' + e - e'' (we remove edge e'' and add edge e to T'). We have that the weight of T'' is less than or equal to T', since $w_{e''} \geq w_e$. But T' is a minimum spanning tree, so the weight of T'' must be equal to that of T'. But we only exchanged one pair of edges, so we must have $w_{e''} = w_e$. This is a contradiction to the statement that all edges in G have distinct weights. Hence G must only have one MST.

DPV 5.7

We negate the edge weights of all edges in G. Then we run Kruskal's algorithm on the modified G. The tree returned is a maximum spanning tree.

KT 4.10

A) We find the shortest path from u to v in T. Adding edge uv to this path forms a cycle. If edge uv is the largest cost edge in this cycle, T is still the MST. Otherwise, T is not the MST.

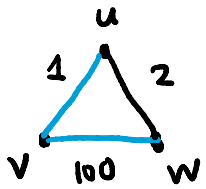
Runtime: Since the graph has a spanning tree, |E| >= |V| - 1. We can find the shortest path from u to v in T using BFS, which takes O(|V| + |E|), which is in O(|E|) time. Checking if edge uv is the largest cost edge takes O(|E|) time. Hence our algorithm runs in O(|E|) time.

Correctness: If edge uv is the maximum-weight edge in the cycle, suppose for sake of contradiction that there is an MST M containing edge uv. Removing this edge from M leaves 2 connected components, one containing u and the other containing v. Since edge uv was part of a cycle, there must be some other path between these two connected components and thus an edge between them. This edge must have cost smaller than edge uv, since edge uv was the maximum-weight edge. We add this edge to produce a smaller MST, which contradicts our previous statement that M was the MST.

If edge uv is not the maximum-weight edge in the cycle, then there is some other edge in the cycle with higher weight. Removing this higher cost edge from T creates two connected components in T. We can add edge uv to T to reconnect the two components. Since edge uv has a smaller cost, this new spanning tree now has smaller weight than our original MST. Hence T cannot be the MST.

B) Let P be the path from u to v in T. We remove the maximum-weight edge on this path, and add edge uv to T to form T'. We prove T' is an MST. T' is a spanning tree, since it is connected and covers all vertices by construction. Consider any edge e' not in T'. If we add e' to T', we obtain a cycle C'. If we add e to T, we obtain a cycle C. e is the maximum-weight edge in C by the cut property. Since the only change from T to T' was an edge swap to an edge of smaller weight, e' is the maximum-weight edge in C', and thus cannot be in an MST. Hence T' is an MST.

CLRS23.2-8 This algorithm does not work. A counterexample is shown below. We partition the vertices into two sets {u} and {v, w}. Then we add edge (v, w) to our result. We unite the two partitions by adding edge (u, v) to the result. This is clearly not an MST.



Interesting problems

DPV5.26 A) (d_1, d_2, d_3, d_4) = (3, 3, 1, 1)

B)

i. Since v_1 has d_1 neighbors and they are not $v_2 \dots v_{d_1+1}$, then v_1 must have some neighbour v_j where $d_1 + 1 < j < n$. Let v_i be some vertex in $v_2 \dots v_{d_1+1}$ (ie $1 < i \leq d_1 + 1$). Then we have that $(v_1, v_j) \in E$ and $(v_1, v_i) \notin E$. Since $i < j$, then $d_i \geq d_j$. Then v_i must have at least one neighbour that is not connected to v_j, let's call it u. In other words, $(u, v_i) \in E$ and $(u, v_j) \notin E$.

ii. We find vertices v_i, v_j, and u as described above. We remove edges (v_1, v_j) and (u, v_i) from E and add edges (v_1, v_i) and (u, v_j) to form graph G'. The degrees of v_1, v_i, v_j, and u all remain unchanged. Hence G' satisfies the degree sequence, and contains edge (v_1, v_i).

iii. We can repeat the above process until the neighbours of v_1 are $v_2 \dots v_{d_1+1}$.

C) Our algorithm works as follows. We first sort the sequence in non-increasing order; let the sorted sequence be $d_1 \dots d_n$. If the degree sequence has length 1: if $d_1 = 0$, return true otherwise return false. Otherwise, we recursively carry out the following procedure. If d_1 < 0 or d_1 > n, we return false. Otherwise, we remove d_1 from the sequence and subtract one from $d_2 \dots d_{d_1+1}$. We can keep the sortedness of the array in d_{max} time.

Runtime: Sorting the sequence takes O(nlogn) time. We have n recursive calls. For each iteration, we perform O(m) decrements. Hence our runtime is O(nlogn + m).

Correctness: We prove correctness of our algorithm by induction on n. Base case: n=1. There is only one node, so d_1 must be 0, which is what our algorithm returns. Inductive step: Assume the algorithm is correct for n=k-1. Consider the degree sequence $d_1 \dots d_k$, sorted in non-increasing order. From part b), we have that if a graph exists with this sequence, there is a graph with this degree sequence with $v_2 \dots v_{d_1+1}$ as neighbors of v_1. We can remove v_1 from the graph to obtain a graph G' with k-1 vertices and degree sequence d_2 - 1… d_{d_1}-1, d_{d_1+1}, . … d_n. By the inductive hypothesis, our algorithm returns a correct result for G'. If such a graph exists, we can re-add v_1 by attaching it to $v_2 \dots v_{d_1+1}$. Otherwise, no graph exists. Hence our algorithm is correct.

CLRS16-1 [Tutorials 5 and 7]
A) We start with the largest coin (quarters) and use as many as possible, then dimes, then nickels, then pennies, until we have a value of n cents. In any optimal solution, the total value of pennies is at most 4 cents. Otherwise you can swap 5 pennies for a nickel. Similarly the total

value of nickels is at most 5 cents, otherwise we can swap 2 nickels for a dime, and so on. Applying this argument to all the coins, we find that the total value of all the coins that are not quarters cannot be more than 25 cents, which means any optimal solution must have the same number of quarters as our greedy solution. We can apply the same argument to the rest of the coins to argue the greedy solution is optimal.

B) Let $x_0 \ldots x_k$ be an optimal solution, where x_i is the number of coins of denomination c^i. We first prove that we must have $x_i < c$ for all i < k. Suppose we have some $x_i < c$. Then, we could decrease x_i by c and increase x_{i+1} by 1. This collection of coins has the same value and has c-1 fewer coins, so the original solution must be non-optimal. To pick as many of the largest coin as possible, the greedy solution results in x_k = floor(nc^{-k}), and for i < k, we have x_i = floor((n mod c^{i+1})c^i). This is the only solution that satisfies the property that there are no more than c of any coin but the largest denomination, since the count amounts are a base c representation of V mod c^k.

C) Coin denominations: (1, 3, 4). n = 6. The greedy solution would output (1, 1, 4), but the optimal solution is (3, 3)

D) We define our subproblems as follows. D(i) is the minimum number of coins to make exactly i cents. We have that D(0) = 0, and D(negative numbers) = infinity. Then we have the recurrence

D(i) = min(D(i - C[j]) + 1) for $1 \leq j \leq k$

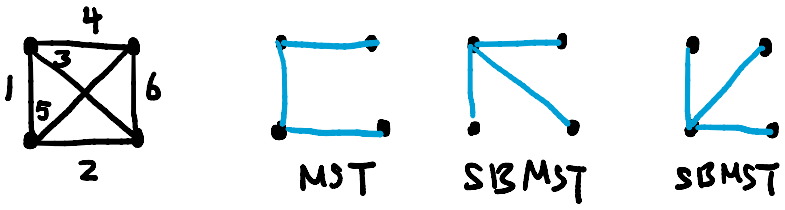
Essentially, to use a coin, we check the minimum number to coins needed to make amount i-coin. Our result is D(n), the minimum number of coins to make n cents.

Runtime: We have n subproblems. For each subproblem, we iterate k times. Hence our runtime is O(nk).

Correctness: We prove the correctness of our algorithm by induction on n. Base case: D(0) = 0, since we need 0 coins to make 0 cents. Inductive step: Assume for all i < j, D(i) is correct. Consider D(j). To make j cents, we must have used at least one coin. We can test one of every possible coin. The number of coins used to make j - coin cents is D(i - coin), which is correct by our inductive hypothesis. Since we take the minimum of all possible coins, D(j) must be correct. Hence our algorithm is correct.

CLRS23-1

A) Minimum spanning tree unique proved above in DPV5.6. An example for second-best minimum spanning tree being not unique is shown below. There is an MST of weight 7 and two second-best MSTs of weight 8.



B) We want to prove that for an MST T of G, we can swap a single pair of edges so that it becomes a second-best MST. Let T' be a second-best MST. Let edge e = (u, v) be in T but not T'.

If we add e to T', we obtain a cycle. One of the edges in the cycle is not in T, let's call it e' = (x, y). We must have that w(x,y) > w(u,v), otherwise we could replace (x, y) in T by (u, v) to obtain a more minimal MST. We can remove edge e' from T' and add edge e to form T''. T'' is a spanning tree, since e and e' are in the same cycle. We also have that the weight of T'' is less than T', so T'' must be an MST. Since G has one unique MST, T'' = T. Thus, T and T' differ only by one edge.

C) Since T is a spanning tree, the BFS tree of T has only tree edges. Let D(u, v) be the maximum edge weight on the path from u to v. D(u, u) = 0 for every vertex u. For every vertex u, we perform BFS to find the maximum weight edge between u and every other vertex. When we visit edge (x, y), we have that D(u, y) = max { D(u, x), w_{xy} }. We also update max(u, v) as we go.

Runtime: T has |V| - 1 edges. Hence |E| < |V|, so BFS takes O(|V|) time. We iterate through each vertex once and perform BFS once on each vertex, so the runtime of our algorithm is O($|V|^2$).

Correctness: We prove correctness of our algorithm by induction on number of edges on a path from u to a vertex v. Let u be an arbitrary vertex. Base case: D(u, u) = 0, since there are no edges on a path from a vertex to itself. Inductive step: Assume D(u, x) is true for all vertices with a u-x path of k-1 edges. Any path with k edges from u to a vertex v must pass through a neighbor of v. Let x be one such neighbor. The path from u to x must have k-1 edges. By the inductive hypothesis, we must have that the maximum weight edge on path u-x is D(u, x). If $w_{xv}$is greater than D(u, x), it must be the maximum weight edge. Otherwise, the maximum weight edge must have value D(u, x). Hence we calculate D(u, v) correctly.

D) We first find the MST of G. Then we compute all max(u, v) as in part c). We iterate over every edge in G\T to find an edge (u, v) that minimizes w_{uv} - $w_{\max(u,v)}$. Our second-best MST is then T with edge max(u, v) removed and edge (u, v) added.

Runtime: It takes O(|E|log|E|) time to find the MST. As in part c), computing all the maximum-weight edges takes O($|V|^2$) time. G\T has O($|V|^2$) edges, so iterating over them all takes O($|V|^2$) time. Hence our algorithm runs in O(|E|log|E| + $|V|^2$) time.

Correctness: From part b), we can replace one edge (u,v) in the MST by another edge (x,y) to get a second-bst MST. If we know which edge (x,y) to add, (u,v) must be the maximum weight edge in the path from x to y. We try every possible edge (x,y) to add and take the best result out of all of them. Hence our algorithm is correct.

CLRS 23-3

A) Let B be a bottleneck spanning tree of G. Suppose for the sake of contradiction that there exists an MST T with an edge (u,v) greater than the weight of B. Let V_1 be the set of vertices reachable from u in T \ v. Let V_2 be the set of vertices reachable from v in T \ u. Consider the cut separating V_1 and V_2. The only edge in T crossing this cut is the one of minimum weight, so (u,v) is the minimum edge crossing this cut. But B has a smaller weight than (u, v). This is a contradiction, since B must have some vertex crossing this cut.

B) We first remove all edges in G with weight greater than b. We select any tree in this modified graph G', such as by creating a BFS or DFS tree. We can test connectivity of this tree by checking that all vertices are visited after the traversal. If the tree spans all vertices, we have a bottleneck spanning tree of value at most b. Otherwise, no such tree exists.

C) [Not covered] We first obtain all the edge weights. We find the median of this list of numbers b and run the procedure in part b) with b. If there exists a bottleneck spanning tree

with weight at most b, we remove all edges with weight greater than b. If there does not exist a bottleneck spanning tree with weight at most b, we contract all edges that have weight at most b. Then we repeat this procedure.

Challenging problems

CLRS16-5 A) WIP