CS 341 - Solutions

Bilal Khan
bilal2vec@gmail.com

April 12, 2023

Many answers copied from Elena's solutions

Contents

1 Cheatsheet **4**
1.1 Master Theorem . . . 4

2 A1 F22 **4**
2.1 Question 1 . . . 4
2.2 Question 2 . . . 5
2.3 Question 3 . . . 5
2.4 Question 4 . . . 5

3 A1 S17 **5**
3.1 Question 1 . . . 5
3.2 Question 2 . . . 5
3.3 Question 3 . . . 6
3.4 Question 4 . . . 6

4 A1 S18 **6**
4.1 Question 1 . . . 6
4.2 Question 2 . . . 6
4.3 Question 3 . . . 6
4.4 Question 4 . . . 7
4.5 Question 5 . . . 7

5 A1 S21 **7**
5.1 Question 1 . . . 7
5.2 Question 2 . . . 7
5.3 Question 3 . . . 7
5.4 Question 4 . . . 7

6 A1 W16 **7**
6.1 Question 1 . . . 7
6.2 Question 2 . . . 7
6.3 Question 3 . . . 8
6.4 Question 4 . . . 8

7 A2 F22 **8**
7.1 Quesrion 1 . . . 8
7.2 Question 2 . . . 8
7.3 Question 3 . . . 8
7.4 Question 4 . . . 9

8 A2 S17 **9**
8.1 Question 1 . . . 9
8.2 Question 2 . . . 10
8.3 Question 3 . . . 10
8.4 Question 4 . . . 10

9 A2 S18 **10**
9.1 Question 1 . . . 10
9.2 Question 2 . . . 10
9.3 Question 3 . . . 10
9.4 Question 4 . . . 10
9.5 Question 5 . . . 10

10 A2 S21 **10**
10.1 Question 1 . . . 10
10.2 Question 2 . . . 11
10.3 Question 3 . . . 11
10.4 Question 4 . . . 11

11 A2 W16 **11**
11.1 Question 1 . . . 11
11.2 Question 2 . . . 11
11.3 Question 3 . . . 11
11.4 Question 4 . . . 11

12 A3 F22 **11**
12.1 Question 1 . . . 11
12.2 Question 2 . . . 12
12.3 Question 3 . . . 12
12.4 Question 4 . . . 12

13 A3 S17 **12**
13.1 Question 1 . . . 12
13.2 Question 2 . . . 12
13.3 Question 3 . . . 12
13.4 Question 4 . . . 12

14 A3 S18 **13**
14.1 Question 1 . . . 13
14.2 Question 2 . . . 13
14.3 Quesrion 3 . . . 13
14.4 Question 4 . . . 13
14.5 Question 5 . . . 13

15 A3 S21 **13**
15.1 Question 1 13
15.2 Question 2 13
15.3 Question 3 13
15.4 Question 4 14

16 A3 W16 **14**
16.1 Question 1 14
16.2 Question 2 14
16.3 Question 3 14
16.4 Question 4 14

17 A4 F22 **14**
17.1 Question 1 14
17.2 Question 2 14
17.3 Question 3 15
17.4 Question 4 15

18 A4 S17 **15**
18.1 Question 1 15
18.2 Question 2 15
18.3 Question 3 15
18.4 Question 4 15
18.5 Question 5 15

19 A4 S18 **16**
19.1 Question 1 16
19.2 Question 2 16
19.3 Question 3 16
19.4 Question 4 16
19.5 Question 5 16

20 A4 S21 **16**
20.1 Question 1 16
20.2 Question 2 16
20.3 Question 3 16
20.4 Question 4 16

21 A4 W16 **16**
21.1 Question 1 16
21.2 Question 2 17
21.3 Question 3 17
21.4 Question 4 17
21.5 Question 5 17

22 A5 F22 **17**
22.1 Question 1 . . . 17
22.2 Question 2 . . . 17
22.3 Question 3 . . . 17
22.4 Question 4 . . . 17

23 A5 s17 **18**
23.1 Question 1 . . . 18
23.2 Question 2 . . . 18
23.3 Question 3 . . . 18
23.4 Question 4 . . . 18
23.5 Question 5 . . . 18
23.6 Question 6 . . . 18

24 A5 S18 **18**
24.1 Question 1 . . . 18
24.2 Question 2 . . . 18
24.3 Question 3 . . . 19
24.4 Question 4 . . . 19
24.5 Question 5 . . . 19

25 A5 S21 **19**
25.1 Question 1 . . . 19
25.2 Question 2 . . . 19
25.3 Question 3 . . . 19
25.4 Question 4 . . . 19
25.5 Question 5 . . . 19

1 Cheatsheet

1.1 Master Theorem

$$T(n) = aT(\frac{n}{b}) + n^c$$

$$\begin{cases} T(n) = \Theta(n^c) & \text{if } \log_b a < c \\ T(n) = \Theta(n^c \log n) & \text{if } \log_b a = c \\ T(n) = \Theta(n^{\log_b a}) & \text{if } \log_b a > c \end{cases}$$

2 A1 F22

2.1 Question 1

1. True, False
2. Depends, Depends
3. True, False
4. True, False

2.2 Question 2

1. $\log \log n$
2. n amount of work at each step, so $n \log \log n$
3. At most constant term larger than $\sqrt{n}$.

2.3 Question 3

2d binary search, find min element in row, recurse in the top/bottom half of the matrix depending on if the min element is $<$ or $>$ its up/down neighbors. This is $O(n \log n)$. Algorithm is correct because if the min element is not a solution, then it is in the top/bottom half of the matrix, and by recursing into the half with a border element that is always smaller than the min element, we will eventually find the solution.

2.4 Question 4

1. If more than half of the chips are bad, then every good chip could be paired with a bad chip, and the test results of that will be inconclusive and we cannot trust the results of any of the chips.
2. recursively pair up chips, remove all pairs that are not good-good. Since there are always more good chips than bad chips, and removing all non good-good pairs will mean we remove all good-bad and bad-good pairs, keeping the invariant that there are more good chips than bad. eventually we reach the base case that there are no bad chips.
3. $T(n) = T(n/2) + 1$. $T(n) = O(\log n) \in O(n)$

3 A1 S17

3.1 Question 1

Programming question N/A

3.2 Question 2

1. $T(n) = T(2n/3) + T(n/3) + n^2$.

$$\begin{aligned} T(n) &\leq 2T(2n/3) + n^2 \\ \log_{3/2} 2 &< 2 \\ T(n) &\in O(n^2) \end{aligned}$$

$$\begin{aligned} T(n) &\geq 2T(n/3) + n^2 \\ \log_3 2 &< 2 \\ T(n) &\in \Omega(n^2) \end{aligned}$$

$$T(n) \in \Theta(n^2)$$

2. size of each level decreases at a rate of $n^{1/2^k}$, so there are $\log \log n$ levels. We do n amount of work at each level, so $T(n) = n \log \log n$.

3.3 Question 3

Problem can be decomposed into, given upper/lower bounds on a rectangle, what should be the left/right bounds of the rectangle to maximize the sum. Iterate over all n^2 upper/lower bounds and for each pair, compute a prefix sum array, summing column-wise, then use kadane's algorithm to find the maximum subarray sum with the left/right bounds in $n \log n$ time. Total time is then $n^3 \log n$.

3.4 Question 4

Recursively build the binary tree given preorder and inorder. preorder[0] is always the root of the tree, so remove that from the preorder array and set that as the value of the node. We can find the index of the root in the inorder array. Count the number of elements n before the root in the inorder array. Split the preorder and inorder arrays into two parts for each of the left and right subtrees by taking the first n elements of the preorder array and the first n elements of the inorder array for the left subtree, and the rest of the elements in both the preorder and inorder arrays for the right subtree. Recurse on the left and right subtrees and add their root nodes as left/right children of the current node. This is correct because the preorder traversal starts at the root node, and the inorder traversal starts at the leftmost node and traverses through all left-children of the root before visiting the root. We can then inductively apply this to each of the left and right subtrees to prove correctness.

4 A1 S18

4.1 Question 1

1. False, $1^0 + 1^1 + 1^2 + \cdots + 1^{n-1} + 1^n = n > c$
2. False
3. True

4.2 Question 2

1. $\log_3 9 = 2$, so $T(n) = \Theta(n^2 \log n)$
2. Total work at level d is $n \log n - 2nd$, and there are $\log n$ levels, so $T(n) = \Theta(n \log^2 n)$
3. Using master theorem and upper/lower bounding it, we get $T(n) = \Theta(n \log n)$
4. $T(n) = O(n \log \log n)$

4.3 Question 3

Outer loop runs $\sqrt(n)$ times, inner loop runs n/j times for each iteration of the outer loop, so total work is $\sum_{j=1}^{\sqrt{n}} n/j = n \sum_{j=1}^{\sqrt{n}} 1/jn \log \sqrt{n} = 1/2n \log n$

4.4 Question 4

Run two-sum on A, storing all possible sums in a hash table (keys=possible sums, values=indices of the two elements that sum to that value). Then run two-sum on B, and for each pair of elements that sum to a value, look up that corresponding value in A's hashtable and return the indices of the four elements if it exists. A's two sum takes $O(n^2)$ time and $O(n^2)$ auxillary memory, and B's two sum takes $O(n^2 \log n)$ time as we need to do a lookup in A's hashtable for each pair of elements in B (assuming binary search lookup time of $O(\log n)$). Total time is $O(n^2 \log n)$.

4.5 Question 5

Programming question N/A

5 A1 S21

5.1 Question 1

Programming question N/A

5.2 Question 2

1. $O(n^2)$
2. $O(n \log \log n)$

5.3 Question 3

Variant of merge sort, split intervals across midpoint, recurse on each half, merge by moving two pointers from mid point and choosing interval with higher height reassigning end time for lower height interval. Total time is $O(n \log n)$.

5.4 Question 4

Choose points for x-start and x-end points for each square, sort by x-start, then iterate through the list and add/remove the y-start and y-end points of that square to an AVL tree. when adding, check if the element before its y-start is a y-start, if the element after its y-end is a y-end (overlapping squares) or if the element after its y-start is before its y-end (overlapping or nested). If so, then there are overlapping squares. Sorting takes $n \log n$, iterating takes $2n$, and there are a constant number of $\log n$ AVL tree operations. Total time is $O(n \log n)$.

6 A1 W16

6.1 Question 1

Programming question N/a

6.2 Question 2

1. $O(n^2)$
2. $O(n \log \log n)$

6.3 Question 3

Sort points by x-coordinate, find inversions in the y-coordinate. This is WLOG equivalent to finding the number of incomparable pairs.

6.4 Question 4

1. Variant of merge sort, split across mid point, recurse on each half and find largest rectangle in each half. Then find largest rectangle that spans across mid point by starting with height of midpoint and decreasing it as you iterate over both halves and multiplying by the width. Total time is $O(n \log n)$.

2. Find max space of rectangle while decreasing baseline height from top of rectangle down to bottom, use a prefix sum array to record current height of each column. Total time is $O(n^2)$.

7 A2 F22

7.1 Quesrion 1

1.
$$5^k n/3^k \sqrt{n/3^k}$$
$$5^k n\sqrt{n} 1/3^k \sqrt{1/3^k}$$
$$(5/3)^k (1/3^{k/2}) n\sqrt{n}$$
$$n\sqrt{n} \sum_{k=1}^{\log n} (5/3)^k 1/3^{k/2}$$

2.
$$6^k (3/7)^{2k} n^{2k}$$

7.2 Question 2

Not doing this one

7.3 Question 3

1. $opt(i,j) = \min_{k \in [i+1,j]}(C(i,k) + opt(k,j))$

2. $n \times 1$ table, Each entry is the minimum cost to travel between city i and city n, The table is initialized to $C[i][j]$, The table will be filled in by iterating over source cities in reverse (n-1 down to 1) and for each of those iterating over intermediate cities (i + 1 to n).

3.
```
ks = []
for i in n-1 down to 1:
    best_k = -1
    for k in i+1 to n:
        if C[i][k] + opt[k][n] < opt[i][n]:
            best_k = k
            opt[i][n] = C[i][k] + opt[k][n]
    ks.append(best_k)
print(ks.reverse())
```

7.4 Question 4

1. The subproblems are knapsack$(i, W_1, W_2) = v$ which is maximum value of items stored in both knapsacks using items in $\{i, \cdots, n\}$ when total weight in knapsack 1 is W_1 and total weight in knapsack 2 is W_2.

$$\text{knapsack}(i, W_1, W_2) = \max_{S_1 \bigcup S_2 \subseteq \{1,\cdots,i\}} \left\{ \sum_{k\in S_1} v_k + \sum_{k \in S_2} v_k | \sum_{k\in S_1} w_k \leq W \sum_{k\in S_2} w_k \leq W \right\}$$

2. Initialize all elements in the table when $W < 0$ to be $-\infty$ and all elements in the table when $i > n$ to be 0.

3. We can observe that at any state we can not add an item to either knapsack, add it to the first, or add it to the second to derive the recurrence:

$$\begin{aligned} \text{knapsack}(i, W, W) = \max\{&\text{knapsack}(i+1, W, W), \\ &v_i + \text{knapsack}(i+1, W - w_i, W), \\ &v_i + \text{knapsack}(i+1, W, W - w_i)\} \end{aligned}$$

4.
```
knapsack[i][W][W] = 0 in all i and W
knapsack[i][W][W] = -inf in all i and W < 0
knapsack[i][W][W] = -1 in all W and i > n
for i in n down to 1:
    for L_1 in 1 to W:
        for L_2 in 1 to W:
            if i == n:
                knapsack[i][L_1][L_2] = w_n
            else:
                knapsack[i][L_1][L_2] = max(knapsack[i + 1][L_1][L_2],
                                            v[i] + knapsack[i + 1][L_1 - w[i]][L_2],
                                            v[i] + knapsack[i + 1][L_1][L_2 - w[i]]
                                        )
return knapsack[1][W][W]
```

5. Not doing, but follows from tracing optimal solution through table w/ aux space.

6. running time is clearly $O(nW^2)$ and space is $O(nW^2)$, but can be reduced by only keeping the last two rows of the table.

8 A2 S17

8.1 Question 1

Programming question N/A

8.2 Question 2

Dijksta, but modified to keep track of the number of shortest paths to each node, incrementing whenever we find a new path to a node with equal length and resetting to num shortest paths of new parent when we find a shorter path.

8.3 Question 3

This is the same as asking can we find a topological sort such that there is a single source vertex. All vertices in a strongly connected component are reachable from each other and so there can be no single source vertex if this is the case. We can fix this by using SCC to find and collapse all SSC into a single vertex and then running topological sort on the new graph. Ofc, if there is more than one component, then there is no single source vertex and we can return false.

8.4 Question 4

Run kosaraju's to find SCC, run the following on each component individually. Odd cycle in directed graph iff graph isn't bipartite, so run BFS and assign colors based on even/odd levels, if you find an edge between two vertices of the same color, then there is an odd cycle that includes that vertex. Print it out by running BFS again from that vertex and keeping track of the parent of each vertex along the way until you reach the same vertex again.

9 A2 S18

9.1 Question 1

Not doing

9.2 Question 2

Recursively find longest path in subtree of left/right child and longest path starting at left/right child (so to pass through root)

9.3 Question 3

Not covered

9.4 Question 4

Schedule by earliest deadline, then by highest value if there are ties. proof by exchange argument from notes :skull:

9.5 Question 5

Not doing

10 A2 S21

10.1 Question 1

Programming question N/A

10.2 Question 2

modified BFS, keeping track of the last three vertices visited and if the current path has three consecutive edges of the same color, set its weight to ∞ to force another path to be taken.

10.3 Question 3

Already done.

10.4 Question 4

Problem of converting an undirected graph into a strongly connected directed graph. Every edge must be included in a cycle and cannot be a bridge, so we can check if this is possible by checking if there are no cut-edges. Modify the cut vertex algorithm and set directions of tree edges from ancestor to descendant and back edges from descendant to ancestor.

11 A2 W16

11.1 Question 1

Programming question N/A

11.2 Question 2

Lol I'm not touching this $O(\min(n^{2k}, n! \cdot n^2))$ runtime.

11.3 Question 3

Already done

11.4 Question 4

Already done

12 A3 F22

12.1 Question 1

The idea is to greedily place every tower as far away from each house as possible in order to cover as much space as possible. We can do this by sorting the houses by their x-coordinate and then placing the first tower at $P[1] + R$. Keep track of the last tower placed and iterate through the rest of the houses. If $|P_i - T_i| > R$, for any house, place a new tower as far away as possible from that house so that it is just covered at location $P_i + R$. The runtime is $O(n \log n)$ and the space is $O(n)$. This is a correct solution as every house is covered by at least one tower and an optimal one as no two towers will have a range that will overlap (since a precondition to placing a tower is that the house is not covered by any tower) and the number of towers is minimal as we place a tower as far away as possible from each house as to maximize the number of future houses that could be covered by that tower.

12.2 Question 2

Not doing

12.3 Question 3

Already Done.

12.4 Question 4

Run Kosaraju's, if more than one component then there is no ultimate vertex. If there is, then topologically sort the graph and run BFS from every source vertex to every sink vertex, keeping track of the number of vertices on the path from the source vertex when visiting a vertex. Reverse the edges and run BFS again and keep the count of the number of vertices on the path from the new source vertex. Vertices with a count of $|V|-1$ are ultimate vertices as every other vertex is reachable from them and they are reachable from every other vertex. This is correct because this is a directed acyclic graph so there are no cycles to count vertices twice and BFS ensures that we visit every vertex exactly once and increment the count along a path exactly once for every vertex on that path.

13 A3 S17

13.1 Question 1

Programming question N/A

13.2 Question 2

They snuck a programming question in here :skull:

13.3 Question 3

Dijksta's but updating thickness of paths so that we check if the minimum thickness along a new path is wider than the current minimum thickness along the current path. Correctness proofs follow that of Dijksta's algorithm. We can print out a path using a parent array.

```
if min(thickness[u], t_{uv}) > thickness[v]:
    thickness[v] = min(thickness[u], t_{uv})
```

13.4 Question 4

Modify the interval coloring algorithm from the notes to make it work in the general case of trying to color as many intervals as possible given k colors. The approach is to sort by non-decreasing finishing time and assign each interval the color of the last colored interval that finished before the start of the current interval, if there exists one. If not, leave the interval uncolored. Use an AVL tree to search for the last interval that ended before the start of the current interval. The runtime is $O(n \log n)$. Correctness proofs follow from the correctness proofs of the interval coloring algorithm by induction.

14 A3 S18

14.1 Question 1

Basically just a modified version of LCS that does comparisons on all m strings instead of just two.

14.2 Question 2

Keep placing books on the current bookshelf until you run out of space and then start a new bookshelf?? proof by exchange, if both books from optimal/greedy solution are on the same shelf we can swap with no change in cost, if they are on different shelves, then we can swap them to add a book to the current shefl.

14.3 Quesrion 3

This is equivalent to the two-knapsack sum problem. Already done. Note we can do knapsack either from n down to 1 or from 1 to n.

14.4 Question 4

DP from n down to 1, setting dp for a location i to true if there exists a word that starts at i and ends at some $j > i$ such that $dp[j]$ is true. n^2 time complexity.

14.5 Question 5

Programming question in disguise

15 A3 S21

15.1 Question 1

Programming question N/A

15.2 Question 2

Greedy solution. Sort jobs by release times. Iterate through release times for each job and add each job to a BST sorted by earliest deadline. Schedule the job with the earliest deadline. If there's a currently running job when a new job is released that we want to schedule, pause it and insert it into its respective place in the deadline-first BST. Runtime is $O(n \log n)$ by obviousness and proof is by exchange, an optimal solution cannot be more optimal than the greedy solution since if there is some part of a job that can run for a while before being preempted in an optimal solution (instead of it being preempted immediately) and still finish by the deadline, we can exchange those time units for the time units of the job with the earliest deadline and still finish all jobs by their deadlines.

15.3 Question 3

Not sure how to do this one.

15.4 Question 4

Check the tree is actually a tree (no cycles, connected) so run DFS on the tree. At the same time check that the shortest path length from the root to a vertex in the tree is $\leq$ the distance to any neighbor of that vertex plus the edge length between the two (to make sure it is in fact a shortest path tree).

16 A3 W16

16.1 Question 1

Programming question N/A

16.2 Question 2

Huffman coding, this is from CS 240

16.3 Question 3

Already done.

16.4 Question 4

Already done.

17 A4 F22

17.1 Question 1

Run Dijkstra's algorithm from every vertex and keep track of the minimum distance to each vertex. Keep track of which s,t pairs have paths in both directions and take the sum of the minimum length paths. Dijkstra's takes $O(|V|+|E|)$ time and there are $O(|V|)$ vertices to run it from, so the runtime is $O(|V|(|V|+|E|))$.

17.2 Question 2

1. Reducing the weight of an edge in a MST will not change the MST. Follows from the cut property of MSTs.

2. Increasing the weight of an edge in a MST will not change the MST.

3. Form a bipartition of the MST by removing e and choosing all the edges in the MST that are now connected to form the bipartition. The cut property states that the lowest-cost edge in the cut of the bipartition will be part of a MST for the graph. We can iterate over all edges in the graph with one end in either side of the bipartition to find the lowest-cost edge in the cut and add that to our MST to complete it in linear time.

4. There is a chance this now reduced-cost edge is part of a MST for the graph. We can check if this is so and if it is, then this edge will be a part of the MST using a similar idea as to what Kruskal's algorithm does (add edges to the MST in increasing order of cost, but only if adding edges does not create a cycle). We would do something in the opposite direction:

add e to the MST, creating a cycle as adding an edge to a tree will always create a cycle, then remove the edge with the highest cost from the cycle. If the edge with the highest cost in the cycle ends up being e, then we can see that T is still a MST after changing the cost of the edge. On the other hand if there is some other highest-cost edge in the cycle e', we can remove that and keep e to form a new MST of lower cost.

17.3 Question 3

Dijkstra's?

17.4 Question 4

Optional, Not relevant anyways

18 A4 S17

18.1 Question 1

Programming question N/A

18.2 Question 2

Sort by increasing height, then apply LIS to the widths. Runtime is $O(n \log n)$ given the fast LIS algorithm.

18.3 Question 3

Programming question in disguise

18.4 Question 4

This is a special case of a min vertex set problem and equivalent to finding a minimum vertex cover in a bipartite graph by bipartitioning the tree on even/odd levels. We can then use the same max flow min cut algorithm to find a minimum cut of the bipartite graph and the vertices in it will be a minimum vertex cover (we need at least one vertex in each edge to be in the .covering set). Ford-Fulkerson takes linear time.

18.5 Question 5

This problem can be reduced to finding a cycle in a directed graph where each weight is non-negative and the product of the weights is positive. We can use the log property of multiplication to reduce the problem to finding a cycle in a directed graph where each weight is non-negative and the sum of the log of the weights is positive. We can further reduce this to the problem of finding a negative cycle in a graph where we further flip the signs of the weights. Bellman-ford is a natural choice for this by running it for n iterations and checking if it finds a negative cycle by not updating the distance of any vertex in the nth iteration. Lastly, we need to run this on each SCC of the graph independently, this takes $O(|V| + |E|)$ time. The graph is represented as a $|V| \times |V|$ matrix where the entry at row i and column j is the weight of the edge from i to j. and so there are $|V|^2$ edges. The runtime is $O(|V|^3)$.

19 A4 S18

19.1 Question 1

Doesn't seem relevant

19.2 Question 2

All conceptual graph theory questions wtf, not relevant.

19.3 Question 3

Dijkstra's lol.

19.4 Question 4

I hate these sort of brainteaser questions. Not relevant.

19.5 Question 5

Ew. programming question. Too lazy to do this and it focues too much on the minutae of the problem to be relevant.

20 A4 S21

20.1 Question 1

Programming question N/A

20.2 Question 2

I'm not sure how to do this one.

20.3 Question 3

Already done.

20.4 Question 4

Dijkstra's with the added constraint of "waiting" at a vertex for the full time of the meeting and minimizing travel time. maximize the number of meetings that can be attended. not sure how to do this :(

21 A4 W16

21.1 Question 1

Programming question N/A

21.2 Question 2

Facility location, already done.

21.3 Question 3

Top-down DP with the following recurrence, choosing the left/right coin that will give the player the best possible winnings once its their turn again after the other player makes the best possible move (the reason why we assume we get the min of the two possible other player results).

$$opt(i,j) = \max\{v_i + \min(opt(i+2,j), opt(i+1,j-1)), v_j + \min(opt(i+1,j-1), opt(i,j-2))\}$$

21.4 Question 4

Already done.

21.5 Question 5

Already done.

22 A5 F22

22.1 Question 1

1. The decision problem is to determine if for some n there are two disjoin subsets of which satisfy the combined value and individual subset weight constraints. This version is in NP as given two disjoint subsets of n items we can check if they satisfy the constraints in polynomial time by just summing up their weights and values.
2. Not really covered in our term, but the idea is about if we knew the problem was solvable for some combined value, it takes polynomial time (really just O(1)) to return the solution (the combined value).
3. Again, just reverse engineering from the values of W_1 and W_2.

22.2 Question 2

Compute cardinality of subset, GCD of subset and original set. easily polynomial time.

Reduce vertex cover to subset gcd by assigning vertives values from the set, and edges between every pair of vertices being the GCD of its endpoints. Then if we choose a subset of vertices to form a vertex cover, the GCD of the values of the vertices in the vertex cover will be the GCD of the original set. Not sure how the backwards direction works...

22.3 Question 3

3-SAT reduction with gadgets not covered.

22.4 Question 4

Approximation algorithms not covered.

23 A5 s17

23.1 Question 1

Convert to d-regular bipartite by adding duplicate vertices, then running d iterations of bipartite matching with Floyd-Fulkerson will give us a schedule that will take at most d days.

23.2 Question 2

Reduction from TSP where the hamiltonian cycle gets edge weights of 1 and all other edges get weight 2. Then it has a hamiltonian cycle iff there is a TSP tour of weight n.

23.3 Question 3

Reduction from Hamiltonian path as a hamiltonian path is a degree-2 bounded spanning tree

23.4 Question 4

Reduction from vertex cover by choosing a set in the set cover to be all edges incident to a vertex in the vertex cover. ez.

23.5 Question 5

Reduction from (directed) Hamiltonian path as choosing all k non-hamiltonian path edges from the graph gives you a subset with at most k edges such that $G - F$ is a (directed) hamiltonian path and therefore acyclic.

23.6 Question 6

This is just finding a maximum bipartite matching using Ford-Fulkerson.

24 A5 S18

24.1 Question 1

1. reducible from clique and IS.
2. reducible from clique.

24.2 Question 2

1. Reducible from vertex cover, each club is an edge, vertex cover is a set of people that are in all clubs.
2. Reducible from subset sum by choosing a subset whos sum is half the total sum.

24.3 Question 3

1. Opposite direction reduction.
2. Shows independent set reduction, not clique reduction.
3. Lol, the size of the clique is bounded at 4 bc the maximum degree is 3 so we can enumerate all n choose 4 4-element subsets in polynomial time.

24.4 Question 4

Not doing, seems overly long.

24.5 Question 5

Not covered.

25 A5 S21

25.1 Question 1

Already done.

25.2 Question 2

Modified version of W23 assignment question, NP-complete, using a reduction from subset sum.

25.3 Question 3

Iteratively set each variable to true then false until the the black box says there is a satisfying assignment for the formula, then set the variable to the value to that true/false value, and remove all (disjunctive) clauses with that variable in it as they short-circuit to true and the truth values of the other variables make no difference. Repeat until all clauses are removed or the black box says there is no satisfying assignment. The proof of correctness follows from this observation and so does the proof of the algorithm being polynomial time.

25.4 Question 4

Already Done

25.5 Question 5

Already Done