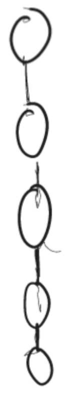
Chapter 2: 2.5.1, 2.5.2, 2.5.8, 2.5.14, 2.5.32, 2.5.33

1. First, we create a stack S, and for each element in a, we will put A[i] to the stack S, and for each element of a stack, we remove the top element and we put it into A[i]. This results in a runtime of O(2n), where we simplify to a big-O time of linear time. For each element in A, we will recurse through the entire array because it has n numbers in order.

2. We now need to reverse the order of numbers in A using the pseudocode where we create a queue, and for each element in A, we will add the element onto the queue. After that, from i equals to length of the array, iterating to Q, we would insert the element into index i of A popping it from Q. This would give us a runtime of O(2n), where we can simplify it to a big-O time of linear time.

8. a) With this problem we need to prove this using induction, where for every node v of T, we have p(v) which is less than or equal to 2 ^ (n+½)-1. Our base case would naturally be when our tree only has one node, where p(1) is less than or equal to 2^(2/2)-1, which gives us one. Our inductive step is on k+1, giving us for some k, p(k) is less than or equal to 2 ^ ((K+1)/2)-1, on p(k+1), we would set it less than or equal to 2 ^ ((K+2)/2)-1, giving us 2*2 ^ ((K+1)/2)-1 = 2*(p(k)). Because this holds by induction on k+1, we can show that it holds on p(v).



8. b) In this case, we would have a binary tree with only five nodes, each connected to the one before it. We have a root, which has one leaf, and this continues three more times. You can take a look at it on the right of this problem.

14. Our goal in this problem is to enumerate all permutations of the set of numbers 1, 2, 3, all the way up to n. In our pseudocode, we input n, where if n is equal to one, we return n, and we end if this is the case. Else, we would recurse on permutations (n-1) + n, giving us the recursive runtime of a big o notation of O(N).

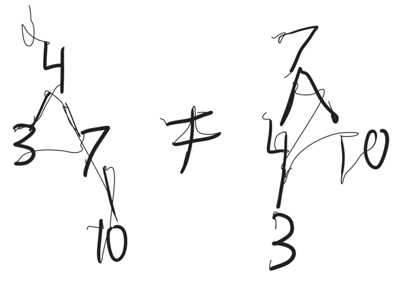
32. In this problem, our goal is to find the lowest common ancestor of two notes, x and y in a tree T. With the root node r, we would start by using depth first search (dfs), where we find the path from the root node to x, and store that path. With the DFC again, we would find the path from r to y, and store that path as well. As such, with the two paths, we could iterate through them and then find the lowest common ancestor of both problems, and then show that the runtime of DFS is linear, looping through the paths takes linear time as well, the overall runtime of the algorithm would be linear.

33. In this problem, we have it arranged as nodes in tree T, where the supervisor of all nodes in the subtree that they are the root of. In order for any two employees to communicate with each other, they need to pass the message for the lowest common supervisor. With us knowing that d is the depth of node v in t, the distance between x and y is the depth of x added to the depth of y minus 2*depth of w, where w is the lowest common supervisor, the diameter of the tree that we have right now, T. In order to find the greatest distance traveled by any message, to find the diameter of T, we would do it using DFS. We can see that the maximum distance will always be between two leaf nodes, and we can pick an arbitrary node to be w, in order to start our problem. Using DRS, we can find that the node furthest from w, is labeled x. We will start the DFS against rom node x, and the furthest node be node y. We

would use the diameter formula, which would give us the greatest distance that the message had to travel. Since we are using DFS, we can calculate the runtime, giving us the runtime in the worst case of the cardinality of V and E.

Chapter 3: 3.5.8, 3.5.18, 3.5.32

8. We would end up with a set of 3, 4, 7 and 10 thus proofing professor Amogus wrong.

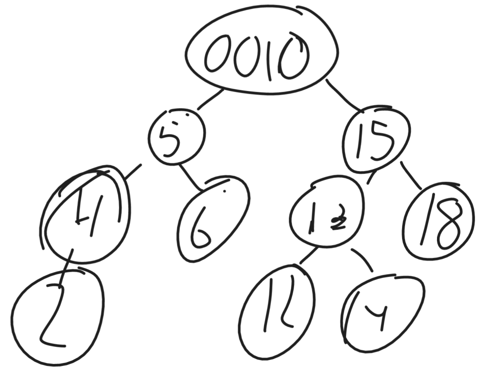


18. The height is at least log(n+1). Our base case is n = 0, where we have no nodes, giving us log(10) = 0, and n = 1, where we have log(1+1) = log(2) = 1. Our inductive step is log(k+1) holds, assuming that 0 is less than or equal to i is less than equal to k holes, because height is equal to one and greater than or equal to holds. On our induction, we need to check that P(K+1) is equal to log (k +2), giving us log(k) + log(2), then by log(k) and log(2), returns our current value by SI, and adding it together is correct because we have k +1 nodes and log(2) is one so it can increase the height by almost 1.

32. For our pseudocode, we need to solve the Josephus problem with n numbers, where we remove every mth number in our problem. We set p = 0, and iterate on n. FOr i = 0, i <n, i + +. Inside, we add m to p, and check if p is greater than n, and set p to n%(p-m), and we end in this case, printing p.

Chapter 4: 4.7.21, 4.7.27, 4.7.42, 4.7.43

21. Assuming that we are in the case where we have an n numbered noded binary tree, we can perform on the left and right rotations, O(n) to convert it to a left chain, or a right chain in our right case. After that, we could have O(n) right and left rotations convert the left chain into any other n node binary tree, and convert the right chain into any other n node binary tree. This gives us an overall runtime of O(n).



27.

42. In this website, we need to add and subtract the dogs on the website, and the most efficient way to do this is to use a binary free. We can create a binary tree that is ordered by the dog the age of each dog, and them we use the red black tree balancing on the left and right rotations and node coloring to remain that the tree remains balanced, because the idea behind using a red black tree, is to have a self balancing tree to save us headaches later down the road. With the runtime of adding a dog, in this case, treated like a node, is O(n), and the runtime of removing a node is O(n) as well.

43. In this case, we need to keep track of our employees in a set, and have them inserted into an AVL tree. We will have employees in a set E, where there is x1, x2, all the way up to xn, each who have their stocks yi, inserting key value pairs of the form of (y, x), where they are employees with their respective promised stocks. The value is some data structure storing employee details like number, age, name, age, years with the company, etc., and the key is the promised stocks. To keep track of the promised stocks, we can have a separate value, Y, which is updated each friday when the stocks are changed, which then changes the values in the AVL tree which is associated with the employee.

Chapter 5: 5.7.8, 5.7.12, 5.7.20, 5.7.27

8. The root holds this value

12. The upper bound is the sum of log(i) is less than or equal to log(n) + … + log(n) = cnlog(n).

20. The explanation is because a heap with n nodes will result with a height of log n, then it makes sense that it would take O(n) in the least case, and best case as O(1), then it is possible for an insertion to be O(nlogn).

27. We need to have two functions, insert which inserts x, a value, to the set S, and then another function called median, which returns the median.

```
insert():
If set = empty
    Median = value
Else
    If value < median
            Insert value in min heap
    Else
            Insert value in maxheap
    If length of min heap is greater than length of maxheap
            Median = minimum pop off
    Else
            Median = maximum pop off
 median();
    Returns the median
```

Chapter 6: 6.6.2, 6.6.10, 6.6.38, 6.6.57, 6.6.74

2. It is omega n^3 because the worst case is omega n^3, and in the best case it is omega n^3. We have three for loops with variables that only increment, so this is why we have an n cubed runtime

10. T(n) = 4 if n=1, and T(n-1) + 4. Our base case is when T(1) is equal to 4, and our inductive hypothesis is that T(k) holds. With our induction, we have T(k+1) equals to T(K+1-1) +4, and T(k) +4, because we know that T(k) holds and returns 4k by the induction hypothesis, then we have 4k+4 = 4(k+1), and our problem returns 4n.

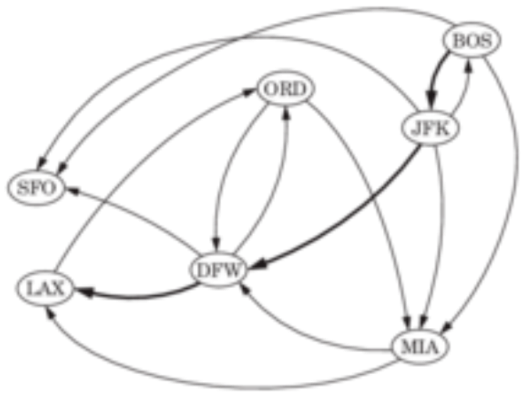
38. T(n) = 1 if n = 0, and T(n-1) + 2^(n-1) otherwise. Our base case is at 0, when it returns 1. Our induction hypothesis is that T(k) holds. Our induction step is that T(k+1) = T(k+1-1) + 2^(k+1), giving us T(k)+2^(k+1). We know that by the induction hypothesis that T(k) returns 2^(k+1) -1 thus we end up with 2^((k+1)+1) which holds, with 2^(n-1) - 1.

57. We would start at row n, and find that from 0, that appears, we get the index-1, and add it to n. Then, we would go to row before at n index-1 and repeat. We end up with this operation where we have O(n) because now go back and start at the beginning of the n rows of any search that has n elements.

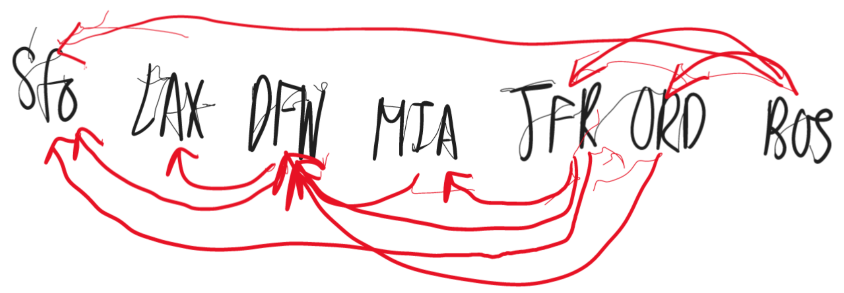
74. We use the hashmap in constant O(1) time, where A is a hashmap, to then go to any, we use V%n which can find the number of occurrences of n that is 1-n in size.

Chapter 7: 7.7.2, 7.7.10, 7.7.25, 7.7.35, 7.7.38

2. It is O(logn) = O(logn) because we have simple connected graph which means that there is at most 1 edge connecting each pair and has no loops, therefore we have m=n thus logm = logn, therefore we end up with the calculation of big o that o(logm) is equal to o(logn).



10.



25. Breadth First Search(BFS) Always produces the shortest path because it searches nodes next before children and takes into account the distance, so for a note to not be visited yet it means it needs to have a greater distance than what was visited before.

35. We would switch to a graph into a divided graph so that every connection becomes a two way connection, then we use either BFS or DFS to run through and get a path that will run in the cardinality of V and cardinality of E time due to the use of either the BFS or DFS algorithm to traverse the graph.

38. The diameter is 2k+1, or 2k. THis is because all the leaves of T repeat this in each iteration, and then in order to find the tree we will have either 1 or 2 noes if 1 to 2k if 2 to 2k+1.

Chapter 8: 8.5.10, 8.5.11, 8.5.14, 8.5.24, 8.5.25

10. Lower bound: 8.4.1, Upper Bound: 8.4.2

11. A v = 100, w=1100, k=1; b v = 50, w = 1

14. We would want to pick the furthest water hole that can be reached and repeat from there. For this algorithm, we would be using the stays ahead proof because it is always going in the direction where we are pitching the wall because of the same or ahead of optimal. Using induction, we know all previous holds and by the definition, we know it will be the one which is farthest possible in each situation.

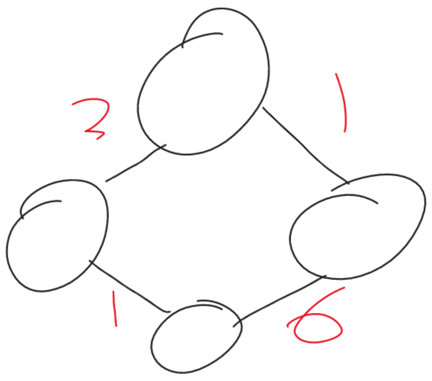
24. We are given a sequence of words where each word has a corresponding length, and we are also given a maximum line length L, and we want to break our words in a way where we are able to minimize the line penalty with each. We have k lines with the indices starting from1 and ending at k, meaning that we want to minimize the whitespace at the end of the lines, and because of this, we would increment through values from W and then append them to the line j, until we cannot fit anymore words into it, meaning that we have successfully minimize the whitespace.

25. Similar to the problem statement from the previous question, we have the same bases, but we will not define the line penalty as something, where we square the penalty. Because we have evaluated the penalty when we have a small, fractional amount of space remaining in the line. When we square the value, we have the algorithm which will think that we have a space when we do not, and therefore our new definition of the line penalty does optimize the line breaking.

Chapter 9: 9.5.3, 9.5.12, 9.5.17, 9.5.18, 9.5.20

3. For this problem statement, we want to modify D’s algorithm to also output a tree T rooted at our starting node, which we will call v, in such a way where we are able to have a vertex, called u, be the shortest path from v to u in graph G. When we run D’s algorithm, we intuitively keep track of the visited nodes, and the nodes are able to keep track of who was their successor and their predecessor in each visit. After we have the algorithm which is done running, we can find the node u in the visited nodes, and be able to trace its path back to v by looking at the nodes which came before it. We can add the path that we find from D’s to the

tree, and then remove all of the node u from the visited nodes, and when we do this, we will have generated the tree T.

12. 

This provides a good counterexample. While the path it takes is from the downwards path, from 1 to 6, we see that the top path is optimal, with an overall cost of 4 rather than 7.

17. We would modify dijikstra's algorithm where it considers edges from left to right that are horizontal.

18. We would make a comparable edge weight where it is greater than 0, and set the rest of the edge weights to 0 so that the dijkstra's algorithm will avoid it.

20. With our airports, we are given them in a set, each of which, A[i], has a connecting time of ct(a). In addition,we are also given a set of flights, where each flight has an airport which it comes from, and a destination airport. As with real life, we want to find a combination where we are able to minimize the time that it takes to fly from one airport to another. We can model this problem by using the model of the shortest path, where we are first using our airports and flights as a graph G(V, E), and then we can start populating our set G with our nodes, with each node corresponding with an airport, because we can store cf(a). In addition, we can connect each node using the flight information. With our new version of dijikstra's shortest path, we can add cf(a) at each node which gives us the most optimal set of flights to get from s to t.

Chapter 10: 10.6.8, 10.6.11, 10.6.22, 10.6.27

8. Because the minimum spanning tree will not revisit a node so even with a weight, a cycle will not be formed in this case.

11. By definition of the minimum spanning tree, it will contain e because it is the lowest cost edge to get to the previous node that node e is connected to.

22. We would modify prim's algorithm to take the largest edge visited instead of smallest.

27. We would use prims, but presort the edges first, and then add them to the priority queue structure in our problem.

Chapter 12: 12.6.1, 12.6.9, 12.6.12, 12.6.16, 12.6.17

1. a) We have a = 2, b = 2, and f(n) = logn. We would end up getting T(n) = Θ(n).

1. b) We have a = 8, b = 2, and f(n) = logn. We would end up getting T(n) = Θ(n^3).

1. c) We have a = 16, b = 2, and f(n) = logn. We would end up getting T(n) = Θ(nlogn)^4.

1. d) We have a = 7, b = 3, and f(n) = logn. We would end up getting T(n) = Θ(n^(log 7(3))logn).

1. e) We have a = 9, b = 3, and f(n) = logn. We would end up getting T(n) = Ω(n^3 logn).

9. With our recurrence relation, we see that T(n) = 2T(n/3)+T(2n/3), T(2) = 1. In this case, we would need to use unrolling to solve this problem. In the second step, we have 2[2T(n/9) + T(2n/9)] + [2T(2n/9) + T(4n/9)], and we would continue this until we get k iterations, where we have 2^kT(n/3^k)+2^(k-1) times kT(2n/3^k)+2^k-2) times kT(4n/3^k) + all the way up to k, which is T (2^(k)n/(3^(k))). For our base case, we end up with log3(n/2), and putting that into our recurrence relation, we will have the asymptotic runtime of O(nlogn).

12. With our set S of pixels in the image, we will divide the set S into new sets set A and set B, where A is the set of pixels which we find inside the bounding box, and then B, a set of pixels which are outside of the bounding box. In order to determine which pixel is in which set, we can use a divide and conquer, where we can, like mergesort, split S into different sets recursively, and then when we reach our base case, in the situation where we only have one element in each set, we would be able to sort them into the different boxes.

16. We would perform a merge sort, after putting our values into different key value pairs in an array, which represents our database. We could have the helper merge function, like we do in every implementation of mergesort, where we have a left, a right, a middle, and when we would be able to swap the middle index and the right index. We would compare in the subarrays, swapping the values when there is less. In order to sort our elements, we would have them into halves, like mergesort does, and then we would break it down until we get to the bottom of the least breakable unit, our singular key value pair. We can merge them using the sorting algorithm, and then we could have the computer which is running in O(n (logn/logk)) time.

17. For each ith interval, each of the intervals within our set fall between 1 and 0, and we have a height, such that c is the greatest height between adjacent buildings. We can use the algorithm by dividing the minimum and maximum height, and have it as a starting point for our pairs. We continue dividing them until we reach our base case, in which we would have constructed the list of pairs, and because we have linear comparisons over a log(n) of cases, we will multiply these two together to end up with an O(nlogn) runtime.

Chapter 13: 13.4.8, 13.4.9, 13.4.28

8. We would use a normal morgue but instead of splitting it into a new array, we would put all the splits in a single array and keep track of the start and end index.

9. We would sort the objects and then check the pairs next to each other to see if they are the same, and then we would then remove and repeat the process again.

28. We can apply numbering to the new alphabet, such that the first letter is the first index (A=1, B=2, C=3, and so on until Z=26) and then convert the letters to numbers, and then we would sort using merge sort.

Chapter 14: 14.5.6, 14.5.8, 14.5.24

6. We can trace through the inductive process to show that using a group of size 3, induction would fail on this problem. In our induction, when we set T(n) is less than or equal to T(n/3 + 1) + T (n/2 + 2) + bn we end up with 5cn/6 + bn + 3c, and then with this following inequality of 5cn/6 + bn + 3c is less than or equal to 5cn/6 + cn/11 + 3c, we end up with 198/5 is less than or equal to n, but we know that 330 is less than or equal o n, which is not fulfilled, so our induction fails on this aspect.

24. We can use pseudocode for an algorithm to solve the problem. The algorithm has only a single for loop, affording us the runtime of linear big-O notation, O(n).

```
Count = 1
Candidate = student
For int i = 2 to n:
    If count is equal to 0
            Candidate = student
            Count = 1
    Else if the array at i is not equal to the candidate
            Count = count -1
    Else
            Count - count + 1
```

Chapter 15: 15.8.10, 15.8.29, 15.8.34, 15.8.35, 15.8.36

10. a) We would use unrolling for this portion of the problem. C(n, n/2) = C(n − 1, n/2 − 1) + C(n − 1, n/2) until our kth iteration, which is C(n-k, n/2− k) + ... + C(n − k, n/2). With k = n/2, and then the first and last terms evaluating to the base cases of C(n,0)=1 and C(n,n), we would have the coefficients following pascal's triangle, and we would end up with a total runtime of at least 2^(n/2).

10. b) By utilizing dynamic programming, we can enhance the efficiency of computing C(n, k), where C represents the binomial coefficient. We can create a solution matrix, denoted as M, with dimensions n × k. The solution for a given input C(n, k) can be obtained from the corresponding entry in the solution matrix, M[n][k]. To populate the solution matrix, we initially assign the value 1 to each M[n][0] and M[n][n], which requires a time complexity of T(2n). The remaining entries of the solution matrix can be filled using the Bellman equation: M[n][k] = M[n - 1][k - 1] + M[n - 1][k] Applying the Bellman equation to fill the matrix takes T(nk) time, resulting in a total runtime of T(2n + nk) = O(nk). Specifically, when considering an input of C(n, ⌈n/2⌉), the asymptotic runtime will be O(n^2).

29. With our string S of n characters, we have a solution matrix M of size n, where M[i] gives us the substring from the beginning to the index i that we would be able to break into different words, in english. We would iterate through the substrings of length i, starting from 1 then incrementing up to n. We would have sij representing the substrings starting from u, and then

all the way up to j. W now have M[i] = valid(sij) $\wedge$ M[j]. If it is true, we can break it down into words, and if it is false, we are unable to. TO find this, we keep track of the indices, and then the S would represent the words that we are able to break it up into. This is n^2 time, iterating through each substring in starting index i in an input.

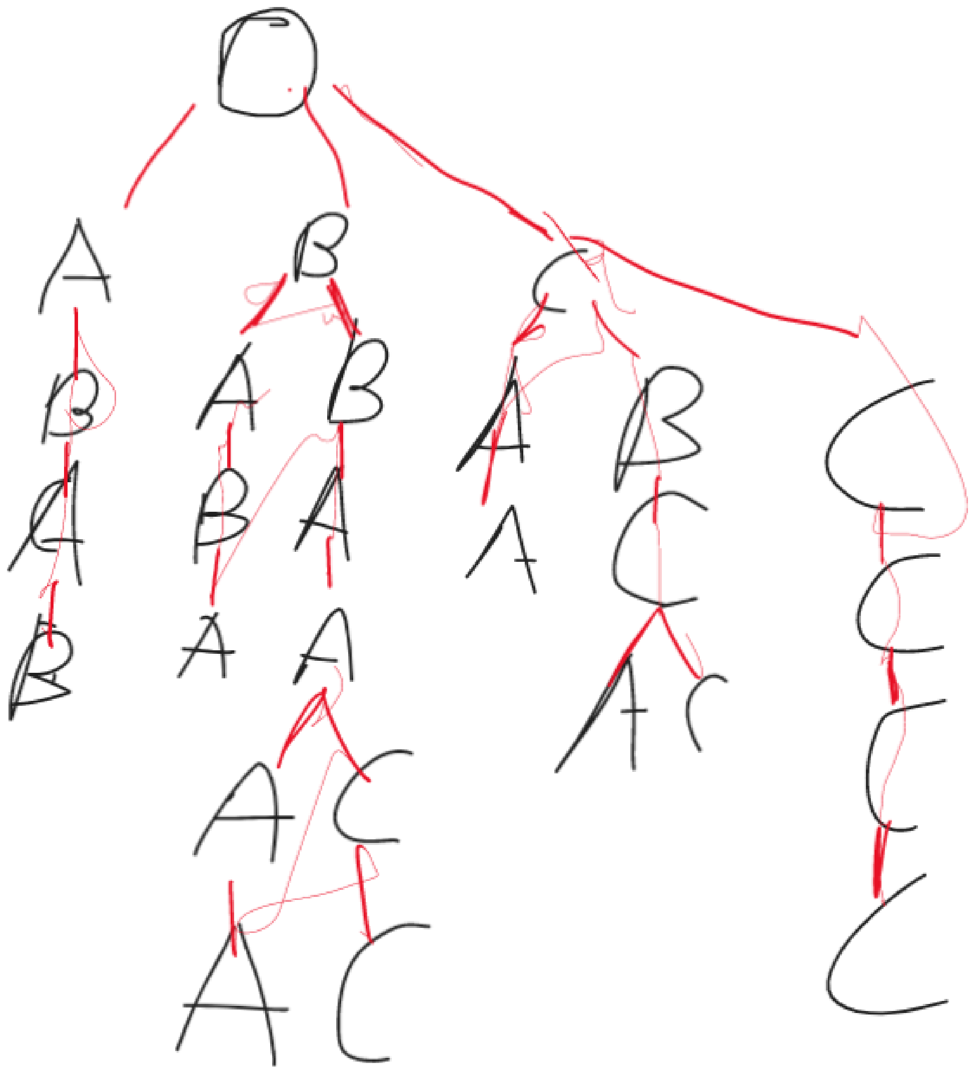
34. In this problem, we are given the numbers representing the streams, as two sets of X, and Y. In the problem statement, we would want to find the mapping while minimizing the distance, and we can have a solution matrix M, where the i and j represents the minimum distance between the first i elements of X, while on the other hand, the j in the second matrix dimension would be representing the first j elements of Y. After initializing 0 0 to 0, and then 0 i, the diagonals, to an infinite weight, we would have 0 j to be infinite as well. Our Bellman equation would end up being M[i][j] = xi − yj   + min{M[i − 1][j], M[i][j − 1], M[i − 1][j − 1]}, and the minimum distance between X and Y would be stored at the bottom left corner of the matrix, and the solution matrix would need O(mn) to be solved.

35. In this problem, we would have a similar coin in a row game, with this time being houses in a row. We would first create a 2d array, dp, of N by , where N is the number of houses in a row, and then we would initialize dp i j with the value of the house at index i if I is equal to j, and if i is not equal to j, we would set the value at 0. We then iterate over dp diagonally, where we move towards the upper right corner. In addition, we would calculate the maximum value that Alice can achieve. When we finish the traversal, the top right element will have the maximum net value. The algorithm has the run time of O(n^2), and then each element can be computed in constant time.

36. To define the parameter V(i, j), which represents the probability that the Anteaters have won i games and the Bears have won j games after playing a total of i+j games, including the special cases V(i,0), V(0,j), V(n/2, j), and V(i, ceiling(n/2)), we can use the following equation: V(i, j) = p * V(i-1, j) + (1-p) * V(i, j-1). We have special cases V(i, 0) = 1, where the Anteaters have won all i games and the Bears have won none. V(0, j) = 0, where the Anteaters have won none and the Bears have won all j games. V(n/2, j) = 0, where the Anteaters have already won more than half of the total required games (n) and cannot lose the series. V(i, ceiling(n/2)) = 1, where the Anteaters have already won more than half of the total required games (n) and cannot lose the series. To determine the overall probability that the Anteaters will win the World Series, we can calculate V(i, j) for all possible combinations of i and j, where i + j = n, and sum up the probabilities where i > j. Mathematically, the overall probability can be expressed as: Overall probability = Σ V(i, j) (where i > j). This algorithm involves filling a table or matrix of size (n+1) x (n+1) and computing the values of V(i, j) iteratively. The time complexity of this algorithm is O(n^2), where n is the total number of games in the series.

Chapter 16: 16.6.8, 16.6.18, 16.6.28

8.



18. We initialize pi0 to 0 and y to 0. We would use the prefix function to find the longest prefix of substring T, maining i and j pointers to point to current positions in T and P, and we would compare the characters in each T and P. If they match, we increment, and then if we don't match, we use the prefix function to update.

28. A highly efficient approach to indexing web pages with different lengths is by utilizing an array of hashtables. Each hashtable within the array is associated with a specific length, where the n-th index corresponds to web pages of length n. The hash function employed can be designed based on the content within the webpage, although the details of the hashing function may vary depending on the web page structure. This data structure ensures a search complexity of O(n) and an additional complexity of O(1), providing optimal efficiency for indexing web pages of varying lengths.

Chapter 19: 19.7.2, 19.7.6, 19.7.28, 19.7.32, 19.7.33, 19.7.34

2. a) Forward edges are v2, v3, v1, v4, to t. The Backward edge is v1 to v3.

2. b) 7 Paths.

2. c) Maximum flow is 12.

6. A min-cut of the graph is the one that separates nodes beta, delta and theta from the sink.

28. In the context of puppy adoption, we can use network flow modeling to optimize the distribution of puppies among Beijing residents. By constructing a graph G = (V, E), we represent residents and puppies as nodes. Each resident node is connected to their preferred puppies, with edge weights set to one. Additionally, a source node (Vsource) is linked to each resident node, while a sink node (Vsink) connects to every puppy node, both with edge weights of one. The maximum flow in this graph reflects the optimal puppy distribution, ensuring residents receive their preferred puppies without exceeding one per resident. However, determining the assignments requires further analysis. By applying depth-first search on the residual graph generated during the Ford-Fulkerson method for finding the max flow, we can identify simple paths from Vsource to Vsink. Each of these paths represents a connection between a resident node and a puppy node, indicating the assignment of a puppy to a resident.

32. The given graph G = (V, E) already represents the problem, eliminating the need for constructing an additional graph. We assign s as the source node and t as the sink node. Since G is a directed acyclic graph, the direction of each edge is known, and there is no requirement to add extra edges to account for bidirectionality. Setting the weight of each edge to one suffices, as overlapping paths would require a flow of two through the edge. The max-flow obtained in the graph corresponds to the number of distinct edge-disjoint paths, any of which can be chosen by the wolf and the goat as long as they do not follow the same path. By performing depth-first search on the residual graph generated during the application of the Ford-Fulkerson method to find the max-flow, we can extract these paths. Whenever the search discovers a simple path with flow from s to t, it is added to a list of paths, and the flow along that path is set to zero. To solve the problem, we simply select two different paths from the list of edge-disjoint paths generated using this approach.

33. To determine the maximum number of creatures that can reach strongholds within their respective regions without overcrowding, we can use a network flow model. Each region (r) has two parameters: Nr, representing the number of creatures in the region, and Sr, indicating the number of strongholds in the region. Each stronghold (s) can accommodate Ns creatures. To model the problem, we construct a graph G = (V, E) as follows: We create nodes (Vs) to represent the strongholds. We create nodes (Vc) to represent the creatures within each region. For region i, we have a creature node (Vci), which is connected to Sri strongholds in region i, represented by nodes (Vsj∈i). The edge weights between Vci and Vsj∈i represent the maximum number of creatures that the corresponding stronghold can hold (Ns). We introduce a source node (Vsource), connected to each creature node, with edge weights corresponding to the number of creatures in the region (Nri). We also include a sink node (Vsink), connected to all the stronghold nodes, with edge weights set to infinity. The flow

into the sink node represents the number of creatures that cannot reach a stronghold. Therefore, the maximum number of creatures that can reach the strongholds is calculated as P i Nri − f in(Vsink), where Nri is the total number of creatures in the region and fin(Vsink) represents the flow into the sink node. By subtracting the flow into the sink from the total number of creatures, we obtain the desired result.

34. We will construct a bipartite graph, with one representing the limos, and then the other set representing the locations. We can connect the nodes with weights based on where the limo is taking the customer, and the weight would be the maximum distance. We add in a sink node with edges leading from location to sink node with a weight of 1. The max flow of this limo graph represents the optimal dispatchment, which we would calculate using FF, giving us the runtime of O(n^2).

Chapter 20: 20.8.1, 20.8.3, 20.8.9, 20.8.24, 20.8.35, 20.8.39, 20.8.41

1. This does not constitute evidence that P = NP. While it is true that P is a subset of NP, and according to the definition, all problems in NP can be reduced to NP-complete problems, it implies that a polynomial time problem L will be reducible to NP-complete, regardless of whether P is equal to NP or not.

3. In order to establish the NP-completeness of SAT, we need to demonstrate two things: first, that SAT belongs to NP, and second, that there exists a polynomial-time reduction from an NP-complete problem Y to SAT. If the certifier for SAT can run in polynomial time, SAT is considered to be a subset of NP. The certificate for SAT consists of a set of boolean input values, denoted as x = x1, x2, ..., xn. The certifier examines each clause and determines its truth value, which can be done in polynomial time, specifically O(n), where n represents the number of clauses.To prove that SAT is NP-complete, we will select CSAT as Y and show that CSAT can be reduced to SAT using an efficient reduction technique. The reduction process is as follows:

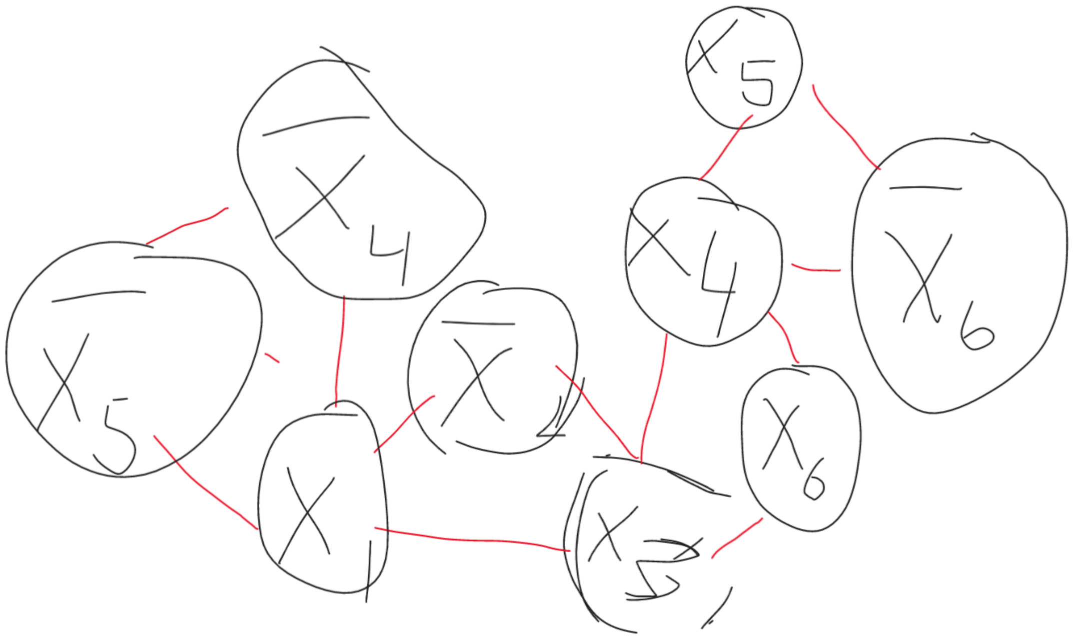
For the NOT gate, xu = xv, we generate two clauses: (xv $\vee$ xu) $\wedge$ (xv $\vee$ xu).

For the OR gate, xu $\vee$ xw = xv, we generate three clauses: (xv $\vee$ xu) $\wedge$ (xv $\vee$ xw) $\wedge$ (xv $\vee$ xu $\vee$ xw).

For the AND gate, xu $\wedge$ xw = xv, we generate three clauses: (xv $\vee$ xu) $\wedge$ (xv $\vee$ xw) $\wedge$ (xv $\vee$ xu $\vee$ xw).

By utilizing a Karp reduction, we can establish that CSAT can be reduced to SAT. If we can demonstrate the equivalence of satisfying assignments between CSAT and SAT (sCSAT $\Leftrightarrow$ sSAT), it confirms the reduction. Firstly, we prove that sCSAT implies sSAT: If sCSAT is a "yes" instance, then the assignment that satisfies the circuit inputs can be utilized to calculate the values of each gate. Through the efficient reduction, these values will satisfy the clauses of sSAT. Secondly, we prove that sCSAT is implied by sSAT: If sSAT is a "yes" instance, then the assignment of variables provides a satisfying assignment for the circuit inputs, and the reduction ensures that the assigned values for the nodes match the gate calculations. Therefore, SAT is NP-complete.

9.



24. Define S to be the set of boolean clauses in a 2SAT problem, we have a G graph with paths P, and we have a corresponding node in G. we want to prove the instance of ST, and it is only corresponding graph G with path from x to not x, and then from not x to x. If not 2SAT, 2AT is not satisfiable, then there is a clause that cannot be satisfied. The clause cannot be satisfied, in our graph G, and we will have a path not x to y, and then from x to y, meaning that not x and x are connected at y, from x to not x and a path from not x to x. Since we know there is a path x to not x, we can assign literal x a value of true or false. Any clause of x or not x will be satisfied, we know that it evaluates to false, and then 2SAT cannot be satisfied.

35. To establish that the job fair decision problem (JF) is NP-complete, we need to demonstrate that JF is in the complexity class NP and that there exists a polynomial-time reduction from an NP-complete problem Y to JF. For JF to be in NP, we can create an efficient certifier that verifies whether a given schedule only includes non competing companies. By checking each company against all others at the fair using a database, we can determine if there are any competing companies. This certifier runs in O(n^2) time, making it efficient and demonstrating that JF is in NP. To establish the reduction from an NP-complete problem Independent Set (IS) to JF, we construct a graph G = (V, E), where each vertex vi represents a company, and edges connect competing companies. This reduction allows us to prove IS ≤p JF. To show the equivalence between instances of IS and JF, we need to prove sIS ⇒ sJF and sIS ⇐ sJF. In sIS ⇒ sJF, if there exists an independent set of size k in G, it implies that there is a set of k non competing companies at the job fair, satisfying the conditions of JF. In sIS ⇐ sJF, if JF is a yes instance, indicating that there are no competing companies at the job fair, it implies that we can construct an independent set of at least size k in G, where k is the number

of companies at the job fair. By demonstrating an efficient reduction between IS and JF, and establishing the equivalence between their instances, we prove that JF is NP-complete.

39. To establish SC's membership in NP, we must show the existence of an efficient certifier. In the case of set cover, the certificate consists of a collection of subsets with a size of at most k. The certifier can iterate over these subsets and verify if their total size is within the specified limit of k. This certifier operates in polynomial time, confirming that SC is in NP. Furthermore, we establish an efficient reduction between SC and the vertex cover problem (VC). By demonstrating that VC can be reduced to SC, we can prove VC ≤p SC using Karp reduction. This reduction implies that sVC (a yes instance of vertex cover) is equivalent to sSC (a yes instance of set cover). To prove the equivalence, we provide two directions: In sVC ⇒ sSC, if we have a vertex cover composed of vertices vr, vr + 1, ..., vk, it implies that the sets Svr, ..., Svk form a set cover, as they cover all the edges incident to vr. In sVC ⇐ sSC, if we have a set cover formed by sets Svr, ..., Svk, where every edge e ∈ E is adjacent to at least one vertex from vr to vk, we can create a vertex cover of size k. By demonstrating an efficient reduction between VC and SC and establishing the equivalence of their instances, we confirm that set cover, and consequently the computer security problem, is NP-complete.

41. The certificate for SP is the ability to partition S into two subsets with equal values. The certifier can partition S into arbitrary sets, sum the values within each set, and compare if the sums are equal. If one of the partitions yields equal sums, it confirms a "yes" instance. The certifier operates in O(n^2) time as there are n different partitions and summing the values takes O(n) time. We choose Subset Sum as the NP-complete problem Y. Subset Sum takes a set of numbers S and a target sum t, seeking to find a subset T ⊆ S such that the sum of values in T is equal to t. For our reduction, we compute the sum of values in S and divide it in half, passing that as the target sum to Subset Sum. By establishing an efficient reduction between Subset Sum and Set Partition, we can prove sSS ≤p sSP using Karp reduction. We need to demonstrate the equivalence of instances, sSS ⇔ sSP: In sSS ⇒ sSP, if Subset Sum has a "yes" instance, indicating the existence of a subset of S with a sum equal to t, it implies the presence of two subsets in S with equal values, as t is exactly half the sum of S. In sSS ⇐ sSP, if S can be partitioned into two subsets with equal values, each subset must have a value equal to half the total sum of S (t). This confirms the existence of a subset in S with a sum equal to t in Subset Sum. By proving an efficient reduction between Subset Sum and Set Partition and establishing the equivalence of their instances, we conclude that the set partition problem is NP-complete.

Chapter 21: 21.10.2, 21.10.18, 21.10.38

2. A = ⅔, B = ⅓, giving us P(x>n/2) = 1-f(n/2), and p(x>(1 +sigma)M) < thus we have r is less than 0.9.

18. With the modification of the FY algorithm, it does not generate each permutation with equal probability, and then if we start with 2, 1, and 3, there is no way we would be able to reach the permutation of 1, 3, and 2.

38. a) Pi = 1/(10^8-i)

38. b) We can evaluate m!/10^8!/(10^8-m)!, bounded by e^(-m^2/2n).

Chapter 23: 23.6.28

28. In our first algorithm, we create two sets and one will be visited and the other will not be visited. In our second algorithm, we connect each city into 2 sets with a path, and then we check the distance. We have X as the last city with double the distance to y. In the end, we would connect both sets.

Chapter 24: 24.6.4, 24.6.13, 24.6.35

4. Q = 2.

13. X1 + x2 is less than or equal to 5.

35. We would use a 5D matrix that has different dimensions for food, water, tent, people, and clothes. We would then use each condition and each element is the number of camels that we used with a solution in the last index.