2y books exercises:

2.5.1

- loop i=1 to n
  push A[i] to a stack S
  end loop
- loop i=1 to n
  pop from stack S & put at A[i] of array
  end loop
- Runtime: O(n) for n elements

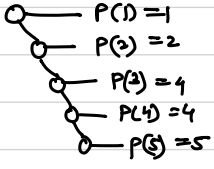
2.5.2

- loop i=1 to n
  Insert A[i] to a queue Q
  end loop
- loop i=n to 1
  Remove from head of Q & put it at A[i]
  end loop
- Runtime: O(n) for n elements.

2.5.8 To find: upper bound for P()

(a) → we can find this when $P(V) \leq n \leq 2^{(n+1)/2} - 1$

(b)



$P(V) \leq V$

2.5.14

```
Permutation (A{1, ..., n})
    if (A{1,...1}, then return 1)
    loop i=1 to n
        Return (i + Permutation ({1,...,n-1})
        Add returned value to output
    end loop
return output set.
```

2.5.32

```
LCA (root, x, y)
    if root == null or root = x or y, return root.
        leftLCA = LCA(root.left, x, y)
        rightLCA = LCA(root.right, x, y)
    if right & left LCA are not null, return root
    if leftLCA is not null, return leftLCA
    if rightLCA is not null, return rightLCA.
end LCA.
```

2.5.33

In order to find the two farthest points,

1) we can run BFS from the root r to find the farthest node
2) we can run BFS from the farthest node found above to find the farthest node from it.

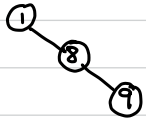
→ we can use these nodes to find diameter distance. We can use the algorithm from the previous question to find LCA to find the distance.

3.5.8

elements : {1, 8, 9}

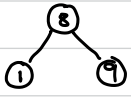
Case 1: order of insertion : 1, 8, 9

tree:



Case 2 : order of insertion: 8, 1, 9

tree:



3.5.18

Base case: P(1)

There is just 1 node, height = 1 & $\lceil \log_2(1+1) \rceil = 1$

∴ P(1) holds.

Inductive hypothesis: Assume P(k) holds for all k nodes.

Inductive step: From inductive hypothesis, we get: for k^{th} nodes, height can only increase by 1.

For $(k+1)^{st}$ node, $\lceil \log(k+1) \rceil \approx \lceil \log(k+1) + 1 \rceil$

∴ P(k+1) holds

Hence, P(n) holds

3.5.32

We can use a circular linkedlist, put a pointer at the head and remove the m^{th} element and store it in a n-sized array. To finish all the elements, this will return n times.

$O(n) \in O(n \log n)$.

4.7.21

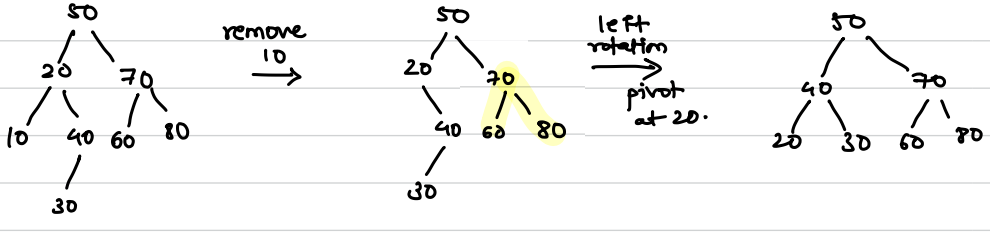
Let us use induction to show that a tree of any height can be converted into a left chain, where each internal node has an external right child node.

Base case: Height 0, there is just one node so it is left chain already.

Inductive step: In a binary tree with n nodes and h height, O(n) rotations can be used on any tree to make it into a left chain.

In a binary tree with height h+1, the binary tree can have two extra children. For 2 children, we will need 2 right rotations to make a left chain on in addition to the O(n) rotations to make the nodes above into a left chain. Thus, induction holds.

4.7.27



so 1 rotation and $\log(8) = 1$

$\Theta(\log n)$.

4.7.42

we can use a doubly linked list. It is O(1) runtime for removing dogs and adding dogs and to find older and newer dogs. we can sort in $O(n \log n)$ time and to put into a doubly linked list, in O(n) time. To find dogs also, we can do it in O(n) time.

4.7.43

Use an AVL tree for efficient $O(\log n)$ insertion/removal and $O(n)$ in order traversal. Addressing the weekly addition of employees involves introducing a unique variable set to 0 initially, adjustable to the CEO - promised value in $O(1)$ time.

5.7.8

In a max heap, the item with the largest key will be the top most node. In a min heap, this would be the bottommost node.

5.7.12

$$\sum_{i=1}^{n} \log i = \log 1 + \log 2 + \log 3 + \dots + \log n$$
$$= \log(1 \cdot 2 \cdot 3 \cdot \dots \cdot n)$$
$$= \log(n!) \leq \log(n^n) = n \log n$$
$$\therefore \sum_{i=1}^{n} \log i \in \Omega(n \log n)$$

5.7.20

Every insertion will need $O(\log n)$ time to finish. Hence, inserting n elements will take $O(n \log n)$ time to complete.

5.7.27

We can use a binary search tree that is balanced. We can use something similar to a quick-select and find the median in $O(\log n)$ time. We can use the size of the balanced binary tree and find the median value in a constant no. of operations $O(\log n)$.

7.7.2

From the question, we know it is a connected graph.

Max no. of edges $= n(n-1)/2$

max value of $m = n(n-1)/2$

$$\log m = \log\left(\frac{n(n-1)}{2}\right)$$

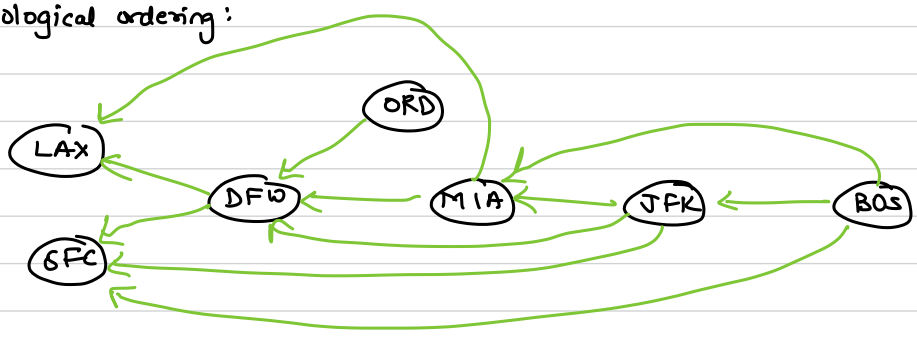
$$= 2\log(n) + \ldots$$

$$\in O(\log n)$$

$\therefore O(\log(m))$ is $O(\log n)$.

7.7.10

topological ordering:



7.7.25

We can use BFS trees, induction and contradiction to prove the property. In the base case where node s and v are both the root r of the BFS tree, the property holds.

Let us assume it holds up to level $(k-1) \rightarrow$ (1)

we consider level k. If there is a path from s to v at level k with fewer than k edges, there is a contradiction. The path in G must have intersected a level j in T less than k, conflicting with v being at level k in T.

Since the property holds for the base case & inductive step, it holds for all cases.

7.7.35

We can use the Euler tour we got for G in O(mtn) time using Hier Holzer's algo for directed graphs which we used for the previous question.
we can then convert the directed edges to undirected edges in O(m) time.

7.7.38

We can run BFS twice to find the diameter of a tree. The first time, we find the farthest node and then we run BFS from that node as the source node again to find the farthest node from this node.
Then, find the distance between the 2 nodes by finding the lowest common ancestor and then running BFS from that node to find the distance.

8.5.10

Every lemma in the Huffman coding algorithm uses an exchange argument for their proof.

8.5.11

Eg: Let each knapsack have space for 1 unit and 2 knapsacks.
Input seq: $< 1/2 + \varepsilon, 1/2 + \varepsilon, 1/2 - \varepsilon, 1/2 + \varepsilon >$
using greedy, biggest first will return in total $1 + 2\varepsilon$, but optimal max is 2 ($\varepsilon << 1/2$).

8.5.14

Algorithm:

At every step of selecting the next water hole, we can sort by increasing distance from the starting point to the water hole. We the choose the water hole with the max distance less than threshold k miles. We do this for every step. This locally optimal solution is globally optimal using a lookahead argument, the distance moved at each step consistently exceeds any alternative, resulting in fewer or equal water holes encountered compared to the optimal solution. Greedy Algorithm is effective.

8.5.24

The greedy heuristic we can use is add as many words to a line as possible and if a word can't fit then break off the last word.

we can use a stays ahead analysis to show that the penalty for this greedy heuristic will always be lesser than or equal to the optimal solution. Since the greedy heuristic fits as many words into a line as possible.

8.5.25

With the changed penalty conditions, limiting each line length to some fixed length L does not ensure minimum penalty. we can have a counter example

Consider string: "xxxxx wwww o ppppp" $L = 5$.

xxxxx

wwww o

ppppp

But the penalty for this line break would be higher than this one ($L^3 > 2$):

xxxxx

wwww

o

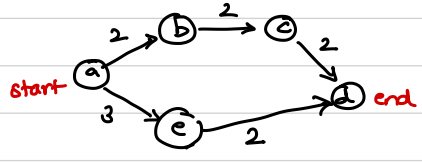
ppppp

9.5.3

In the Dijkstra code, just before a node has been added to the set S, note down its corresponding parent too. Start tree with v, then after the whole code has finished, make all nodes with v as the parent as the children of v in the tree, do this for each parent-child relationship to get the tree.

9.5.12

counter-example:



According to the algorithm: shortest path: $a \to b \to c \to d$

total cost : 6

optimal solution: $a \to e \to d$, total cost: 5.

9.5.17

The conventional Bellman-Ford algorithm should work here. We just need to make sure the left-right constraint has been checked. Complexity: $O(n^2)$, because we need to consider all pairs of vertices. we would just need a one dimensional solution matrix A where A[i] would represent the min cost of a monotone path from our starting point to node i.

9.5.18

We can transform the problem into a directed graph $G = (V, E)$ where V = stations and $|V| = n$ and E = communication channels and $|E| = m$. cost of compromised edges = 2. Cost of uncompromised edge = 1.

Run Dijkstra's. Min cost = lowest no. of compromised stations.

9.5.20

Create a graph with airports as nodes and directed edges from origin to destination airports. Use a 1D matrix A where A[i] represents the min time to reach airport i. Pre-processing time data balance before running the Bellman-Ford algorithm, which overall takes $O(n^3)$ time.

10.6.8

For prim's, edge added will always be the least weight edge so it doesn't matter if its a negative edge.

For Kruskal's, the min edge is also considered and then we add it to the tree. Below is a modification of the Prim's algorithm, hence the same reasoning applies. None of the algorithms are affected by min weight cycles because all of them create trees and for every tree, the no of nodes is 1 more than the edges so a cycle can never be a part of the tree.

Kruskal avoids adding cycles between already-connected nodes. Prim's and Belonuvls use a priority queue, preventing the addition of edge to visited nodes and avoiding negative weight cycle impact.

10.6.11

We can use the cut property. Let $e = (v, w)$ be the edge. Create a cut with $S = \{v\}$ and $V-S$ would then have all the other edges. e is the edge with the min weight here so it has to be in the MST.

10.6.22

Use Prim's. Start with an arbitrary edge. Use a priority queue but one that starts by decreasing then increasing order. Time: $(|E| \cdot \log |E|)$.

10.6.27

We create a graph with each dorm as one of the n nodes and the m possible connections on the edges. These edges will have weights either 20 or 50 based on the distance between the nodes according to the question. Run prim's normally to get the connection graph which takes the min cost.

12.6.1

According to the master theorem, the recurrence relation would be:

a) $T(n) = O(n)$

b) $T(n) = O(n^3)$

c) $T(n) = O(n^4 (\log n)^5)$

d) $T(n) = O(n^{\log_3 7})$

e) $T(n) = O(n^3 \log n)$

12.6.9

We can use mathematical induction to prove the algorithm's correctness. It is shown to work for 1 and 2 element arrays in the base case.

For the inductive step, assuming it works for smaller arrays, we demonstrate its effectiveness for $A[i, \ldots j]$. The three recursive calls return correct values by the inductive hypothesis.

After sorting all these parts, the entire array $A[i, \ldots j]$ is "Stooge-sorted".

The recursive running time is $T(n) = 3T(2n/3) + O(1)$ and by the master theorem, $T(n) = O(n)$.

12.6.12

we employ a divide and conquer algorithm that recursively splits co-ordinates into two halves, returning their min and max values. Linear time comparision of these values yields the answer. The base case handles sets with a single element, while for larger sets, we recursively compute min and max, combining them to find the overall min and max.

Recursive Definition: $T(n) = 2T(n/2) + O(1)$ for fewer than $3n/2$ comparisions.

By the master theorem : $T(n) = \theta(n)$

12.6.16

Performing k-way merges with magic register, akin to merge sort, we can sort k-length sublists in linear $O(n)$ time. The merging of these k-sublists groups takes k^2 time. we can then create lists of length k^3, k^4, etc.

This sorting, taking $O(n)$ for each level for each $O(N)$ levels, results in runtime of $O\left(\frac{N \log N}{\log e}\right)$.

12.6.17

We use a divide and conquer method, similar to Mergesort. We compare the x-co-ordinates of two skylines at a time, adding the one with the smaller x-coordinate to the result. the height is set to the maximum of the 2. This process has a time complexity of $O(n \log n)$.

13.4.8

The variation of the merge-sort can be implemented using a bottom up approach.

1) Divide the input array into n sub-arrays, each size 1.
2) Merge adjacent pairs and store in T.
3) Repeat above step but this time merge arrays of size 2.
4) Keep following the above heuristic while using S and T alternate destination arrays, until you reach one merged array of input size n.

13.4.9

Sort objects in $O(n \log n)$ time. Then linearly go through array, removing duplicates, thus creating a set. Runtime: $O(n) + O(n \log n) = O(n \log n)$.

13.4.28

Perform Radix Sort. we would have m bucket, we have to perform some constant no. of iterations based on the upper bound of the length of the name and sorting in each iteration can be done in $O(n)$ time based on the ordering of letters. Runtime: $O(mn)$.

14.5.6

The runtime for deterministic section depends on the group sizes. It is only the group size of 5 that guarantees that at least 3/10 of the elements are greater than the median of medians. The group sizes allow us to bound the size of the sub-problem. Taking a smaller group size would not guarantee our logic. Group size of 3 would only guarantee 1/3 of the elements being greater than the MOM.

14.5.8

We can make any comparision-based sorting algorithm by changing how elements are compared. Instead of using an operation like $>$ or $<$, we can use a composite key which checks the position of the elements in the ordering. This does not change the asymptotic time of the algorithm while also making the algorithm stable.

14.5.24

Use an array with each of the candidates initialized to 0. Go through the n votes, and stop counting if, while incrementing votes, you find a candidate which has more than $n/2$ of the votes. If the program has to go through the whole array then no candidate got majority vote.

Runtime: $O(n)$.

15.8.10

Recursive Function: $C(n,k) = C(n-1, k-1) + C(n-1, k)$ for $0 < k < n$

Base case: $C(n,0) = 1$

$C(n,n) = 1$

a) To find: $C(n, n/2) = C(n-1, (n/2)-1) + C(n-1, n/2)$

Since $k < n$, $k-1 < n-1$, so both recursive cases will go to a simpler case until a base case is reached.

For an even n, the $(n/2)$ value goes to $(n/2)-1$ after each iteration. So we have atleast $n/2$ levels with each level having twice the no. of recursive calls as the level below so atleast $2^{n/2}$ recursive calls so atleast $2^{n/2}$ runtime.

b) Have a solution matrix of $n \times k$: start from the base cases, compute values row-by-row. Since $k < n$, value will be at $[k][n]$ of matrix with $O(n * k)$ runtime

15.8.29

We can break the string S into valid words in multiple ways. One approach is to use memorization with a tree for finding all possible word combinations in $O(nk)$ time where k is the max word length.

Another option is to use n^2 array. Starting from (0,0) and (1,1) we populate the array to store whether each substring forms a meaningful word. This matrix allows us to find all possible combinations of contiguous valued words in n^2 time.

15.8.34

Create a matrix $M[n+1][m+1]$

Initialize $[0,0]$ to 0.

$M[i][i] = d(x[i], y[j]) + \min(M[i-1][j], M[i][j-1], M[i-1][j-1])$

Return $[n][m]$ value.

Runtime : $O(mn)$.

15.8.35

Let $1 \ldots n$ be all hours in a row. We will use a 2-d solution matrix A, which is $n \times n$ in size.

Let $A[i][j]$ be the max net value which Alice can get if Bob plays optimally afterwards

$$\text{AliceOPT}[i][j] = \max \begin{cases} \text{BOBOPT}[i+1][j] + \text{cost}(i) \\ \text{BOBOPT}[i][j-1] + \text{cost}(j) \end{cases}$$

$$\text{BOBOPT} = \min \begin{cases} \sum \text{cost}(k) + \text{AliceOPT}[i+k][j] \\ \sum \text{cost}(k) + \text{AliceOPT}[i][j-k] \end{cases}$$

Base case: fill in all the diagonals with value 0.

Populate the matrix row by row, top to bottom and answer will be the max of all the values in the top most row.

15.8.36

$V(i,j) = P * V(i-1,j) + (1-P) * V(i,j-1)$

$V(i,0) = 1$, for all i

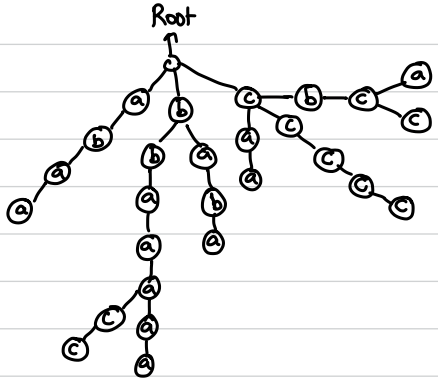
$V(0,j) = 0$, for all j.

$V(\lceil n/2 \rceil, j) = 0$ for $j \neq \lceil n/2 \rceil$

$V(i, \lceil n/2 \rceil) = 1$ for $i \neq \lceil n/2 \rceil$

$P(\text{Anteaters win world series}) = \sum_{i=1}^{(n+1/2)^{th}} V(i, n-i)$

16.6.8



16.6.18

We can just ask the Knuth-Morris-Pratt algorithm which will find solution in $O(m+n)$ time.

16.6.28

The web crawler can use a hash-table to achieve this. For a web-page of length n, it will take $O(n)$ time to compute. Value of $O(1)$ to know if website has been encountered before and $O(1)$ to add website if it is new.

23.6.28

To show NN Greedy Approach is a 2 approx-algo for metric TSP, we need to show its tour length is at-most twice the optimal length.

T_G: tour by Greedy Algorithm

T_O: optimal solution.

Let $c_1, c_2, \ldots, c_n$ be cities in the order added to T_G and let d_i be the city in T_O that follows c_i.

$$\begin{aligned} d(T_G) &= \text{sum of distances between consecutive cities in } T_G. \\ &= d(c_1, c_2) + d(c_2, c_3) + \cdots + d(c_{n-1}, c_n) + d(c_n, c_1) \\ &\geq d(c_1, d_1) + d(d_1, c_2) + d(c_2, d_2) + \cdots + d(c_{n-1}, d_{n-1}) \\ &= d(T_O)/2 \end{aligned}$$

The inequality above holds because at each step, we choose the nearest city not in T_G, so $d_i \leq d(c_i, d_i)\ \forall i$

$\therefore\ d(T_n) \leq 2 \cdot d(T_O)$

24.6.4

To determine if the problem has no unique solution, we need to check the constraints and the objective function. If the constants are redundant or the objective function is parallel to one of the constraints, then the problem has no unique solution.

slope of objective function = a

slope of first constant = $-3/5$ $\Rightarrow$ not parallel.

So, the problem has no unique solution for all values of alpha and there is no values of alpha that results in a program with no. unique solution.

24.6.13

Let the dual variables be a_1, a_2, a_3, a_4 for the four constraints in the original linear programming.

$\Rightarrow$ dual linear program is: minimize : $5a_1 + 3a_2 + 24a_3 + 9a_4$

subject to : $a_1 + 6a_2 + 5a_3 \geq 1$

$a_1 - 3a_2 \geq 2$

$a_3 \geq 0$

$a_4 \geq 0$

minimize : $w = 5a_1 + 3a_2 + 24a_3 + 9a_4$

subject to the constraints above.

24.6.35

The linear program given the different constraints can be formalized as

minimize : $75x_6 + 50x_7 + 25x_8 + x_1 + 2x_2 + x_3 + x_4 + x_5$

subject to : $2x_1 \geq 25$

$2x_2 \geq 7.5$

$x_3 \geq 22.5$

$x_4 \geq 18.75$

$x_5 \geq 15$

and $18x_1 + 8x_2 + x_3 + 10x_4 + 20x_5 + 75x_6 + 50x_7 + 25x_8 \leq 800n$

n: no. of camels.

21.10.2

Let m be the no. of games Beans win out of the n games.

To find: The probability that Beans wins a majority of the games $\Rightarrow [P(m > n/2)]$

we can use a chernoff bound for this:

The upper bound on the probability for $\delta = 1/2$ using chernoff is $< 0.9^n$.

21.10.18

The case n=3 gives a good counter-example to show that the given algorithm doesn't perform every permutation with equal probability.

The permutation [1,2,3] is generated with possibility 1/3 while the other two permutations [1,3,2] and [2,1,3] are generated with a 1/6 probability. Because it is the first step of the algorithm, there are n possible swaps to choose from and each 1 has probability of 1/n, however the array is no longer symmetric after the first swap. So the possibility of generating certain permutations is higher than others.

21.10.38

a) As probability that i^{th} employee has a distinct pin is $(n-i+1/n)$. Since there are $(n-i+1)$ possible PINs that the i^{th} employee can choose.

b) Probability of all m PINs being distinct = Product of Pi's $(i = 1, 2, \ldots, m)$

Since $1 - x \leq e^{(-x)}$ for $0 < x < 1$

$\Rightarrow \frac{n-i+1}{n} \leq e^{-i-1/n} \Rightarrow P_i \leq e^{(-(i-1)/n)}$

So, the product of P_i is bounded by $e^{-1/n} \times e^{-2/n} \times \cdots = e^{\frac{-(-(1+2+\cdots+m+1))}{n}}$

c) we want to know the probability of two employees that have the same PIN $\geq 1/2$.

so, we want $1/2 \leq e^{\left(\frac{(n)^2 m^2}{n}\right)}$

taking log,

$\ln(1/2) \leq -\frac{1}{2} \times \frac{m^2}{n}$.

When we solve for n, we get that the system has to produce at least $m^2/2 \ln(2)$ possible PINs to ensure probability $\geq 1/2$.

19.7.2

Forward edges : (s, v_2), (v_2, v_3), (v_1, v_4)

Backward edges : (v_1, v_3).

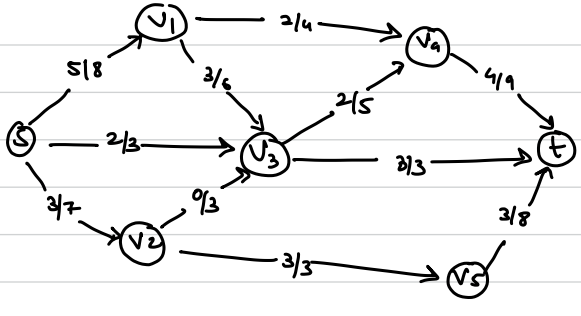
Augmenting path: $s \to v_2 \to v_3 \to v_4 \to t$ $(c=3)$

$s \to v_1 \to v_4 \to t$ $(c=2)$

$s \to v_1 \to v_3 \to v_4 \to t$ $(c=3)$

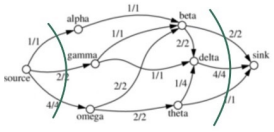
$s \to v_3 \to v_4 \to t$ $(c=1)$

Max flow : 15



19.7.6

possible min cuts!



(h)

19.7.28

m and n become two sets of nodes: an edge between these two sets of nodes if resident $a \in n$ likes puppy $b \in m$ an edge from source node s to each node $\in m$ and an edge from every node $\in n$ to sink node t. Each edge has capacity 1 and we only allow integer flow.

max flow: max. no. of puppies allowed.

we can find matching between a person and an edge with flow 1 represents one matching.

19.7.32

Make every edge have capacity 1 with only integer flow allowed. If max flow ≥ 2, there do exist two distinct paths for the wall and goat.

Path Decomposition

- Let f be a max-flow for this problem. How can we recover the k edge-disjoint paths?
- DFS from s in f along edges e, where $f(e) = 1$:
  1. Find a simple path P from s to t: set flow to 0 along P; continue DFS from s.
  2. Find a path P with a cycle C before reaching t: set flow to 0 along C; continue DFS from start of cycle.

to recover paths which wall and goat can use.

19.7.33

Nodes : Region r and stronghold s.

Edge : From region r to every stronghold in set Sr.

Super : sources connected to all of the regions and super sink t connected to all of the strongholds.

capacity s to region r is Nr of capacity from stronghold s to t in Ns.

Capacity between regions and strongholds can be Infinite.

Max-flow : max no. of creatures we can remove.

19.7.34

Nodes : n limousines and n locations.

Edge : From source node s to all limousines and edge from all n locations to sink node t

Capacity : for all of these edges is ∞.

An edge from limousine $a \in n$ to $b \in n$ locations where capacity is dab! min cut minimizes distance.

20.8.1

It does not solve $NP = P$ because the problem L is being reduced to an NP-complete problem instead of the other way around.

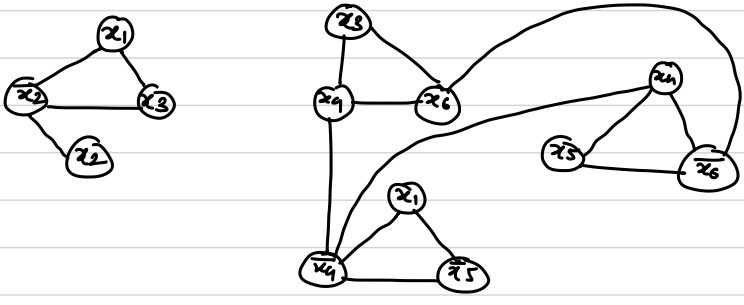
20.8.3

We demostrate the NP nature of the problem by establishing an efficient certifier, validating solution in polynomial time $(O|s|)$.

TO prove NP − correctness, we reduce the problem to CSAT, a known NP-complete problem, by introducing an and operator between clauses.

The time complexity remains polynomial, confirming the problem as NP- complete.

20.8.9



20.8.24

Assuming a 2SAT problem instance represented by graph G is unsatisfiable, we can determine this by checking if there is a path from a variable x to its negation $\sim x$ and vice-versa. If such a path exists, it implies a contradiction and renders the 2SAT problem unsatisfiable.

This check can be efficiently performed using Kosarajus algorithm in $O(n+n)$ time.

20.8.35

To establish the NP-correctness of the problem, we first demonstrate its membership in NP by presenting an efficient certifier that checks a solution sequence in $O(n^2)$ time. Having confirmed its verifiability in polynomial time, we can prove NP-completeness by polynomial time reduction from a known NP-complete problem, such as 3SAT.

By associating each variable in 3SAT with their corresponding companies in our problem, and ensuring satisfiability implies a specific company selection, we get a connection between the two problems.

20.8.39

To show that the problem is NP-complete, we need to first show that it is NP. We can do this by showing it has an efficient certifier. Given a solution sequence of the websites which all the infected computers have visited, we can check if their list is correct in polynomial time down to some constant factor $\uparrow$ # of computers by checking each website with each log file of infected computer.

Now, to show NP-completeness, we can reduce an already NP-complete problem to this problem in polynomial time and show an if and only if correlation between them.

Let us show set cover $<=_p$ our given problem. we will have one computer per element and one site per set. Then, for a set S_i corresponding to it and put it in the log of computers in that set. Then, if there are n suspicious sites, there would be n sets in the set cover. If there are n sets, then there are n sites. By this reasoning, we can show that the problem is NP-complete

20.8.41

To show that the problem is NP-complete, we need to first show that it is NP. We can do this by showing it has an efficient certifier. Given a solution sequence, we can check if the sum of the sets in the two positions are equal which is an implication that the solution sequence is a yes-instance. Since we can check the solution sequence is polynomial time $O(n)$ where n is number of books in each set, we have an efficient certifier and the problem is NP.

Now, to show NP-completeness, we can reduce an already NP-complete problem to this problem in polynomial time and show an if and only if correlation between them.

Let us show set cover $\leq_P$ our given problem. We can take all elements in the set and assign the books and have their value as the books cost. Then there would be a valid partition if and only if there is a valid split of the books so the problem is NP-complete

6.6.2

Algorithm : MaxSubSlow(A):

Input : An n-element array A of numbers, indexed from 1 to n.
Output : The max subarray sum of array A

```
m ← 0
for j ← 1 to n do
    for k ← j to n do
        s ← 0
        for i ← j to k do
            s ← s + A[i]
        if s > m then
            m ← s
return m
```

max $\frac{n(n-1)}{2}$ iterations

runs max of n iterations.

Runtime is : $n \cdot n \cdot \frac{n-1}{2} \in \Omega(n^3)$

6.6.10

$$T(n) = \begin{cases} 4 & \text{if } n=1 \\ T(n-1)+4 & \text{otherwise} \end{cases}$$

Base case: $T(1) = 4$, $4 \cdot 1 = 4$ } LHS = RHS

$\therefore$ Base case holds.

Inductive step: Assume $T(K) = 4K$ holds true for some $K \in N^+$

Now, showing $T(K+1) = 4(K+1)$ holds true.

$T(K) = 4K \longrightarrow$ IH

$T(K+1) = T(K+1-1) + 4$

$= 4K + 4$

$= 4(K+1)$

□

$\therefore$ Induction holds and $T(n) = 4n \quad \forall n \in N$

6.6.22

$$T(n) = \begin{cases} 1 & , \text{ if } n=0 \\ T(n-1)+2^n & , \text{ otherwise} \end{cases}$$

To prove: $T(n) = 2^{n+1} - 1$

Base case: $T(0) = 1 \quad [n=0]$, and $2^{0+1} - 1 = 1$ } LHS = RHS. Base case holds.

Inductive step: Assume $T(k) = 2^{k+1} - 1 \longrightarrow (IH)$

Now showing $T(k+1) = 2^{k+2} - 1$

$$T(k+1) = T(k+1-1) + 2^{k+1}$$
$$= T(k) + 2^{k+1}$$
$$= 2^{k+1} - 1 + 2^{k+1}$$
$$= 2 \cdot 2^{k+1} - 1$$
$$= 2^{k+2} - 1$$
$$= 2^{(k+1)-1} = RHS$$

By induction, $T(n) = 2^{n+1} - 1$

6.6.57

Algorithm: Go through the array row by row.

1) Create a variable "prevcount" to note down the number of 1's found in the previous row. Initialize this to 0.

2) When first 0 is encountered in any array, update this in total count and update prevcount to position of first 0 found.

3) In next row, start counting from position prevcount until first 0 found. Update number of 1's found + prevcount value to total count.

Since we are skipping over the array cells which have 1 in each row, whenever we move onto the next row, the total array accesses will stay $O(n)$ and since we are updating prevcount and totalcount once per every row, these operations all combined, are also $O(n)$. So, total runtime is $O(n)$.

6.6.74

Sum on numbers from 1 to n is $\frac{n(n+1)}{2}$.

→ Go through array and compute the following things in each cell and store in variables.

1) Running total of elements (O(1)).
2) Running total of square of elements.

Formula for sum of squares from 1 to n is $\frac{n(n+1)(2n+1)}{6}$.

Now, we have a system of equations to find two numbers. x x y.

→ $x+y$ = total sum till n - running total

→ $x^2 + y^2$ = total no. of squares.

These system of eq. can be solved in O(1) time.

So, O(n) runtime of O(1) space to find x and y values.