19. You're consulting for a group of people (who would prefer not to be mentioned here by name) whose jobs consist of monitoring and analyzing electronic signals coming from ships in coastal Atlantic waters. They want a fast algorithm for a basic primitive that arises frequently: "untangling" a superposition of two known signals. Specifically, they're picturing a situation in which each of two ships is emitting a short sequence of 0s and 1s over and over, and they want to make sure that the signal they're hearing is simply an *interleaving* of these two emissions, with nothing extra added in.

This describes the whole problem; we can make it a little more explicit as follows. Given a string x consisting of 0s and 1s, we write x^k to denote k copies of x concatenated together. We say that a string x' is a *repetition* of x if it is a prefix of x^k for some number k. So $x' = 10110110110$ is a repetition of $x = 101$.

We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two (not necessarily contiguous) subsequences s' and s'', so that s' is a repetition of x and s'' is a repetition of y. (So each symbol in s must belong to exactly one of s' or s''.) For example, if $x = 101$ and $y = 00$, then $s = 100010101$ is an interleaving of x and y, since characters 1,2,5,7,8,9 form 101101—a repetition of x—and the remaining characters 3,4,6 form 000—a repetition of y.

In terms of our application, x and y are the repeating sequences from the two ships, and s is the signal we're listening to: We want to make sure s "unravels" into simple repetitions of x and y. Give an efficient algorithm that takes strings s, x, and y and decides if s is an interleaving of x and y.

Given:

- An interleaved string s with n total characters.
- Two strings x' and y' (repetitions of x and y) that also have exactly n characters each.

We want to: Determine if s is an interleaving of x' and y' .

Algorithm:

1. Iteration through s :
    - Iterate through the string s to match characters with x' and y' .
    - If a character in s matches with x' , move the pointer in both x' and s forward by one character.

- Similarly, if a character in s matches with y' , move the pointer in both y' and s forward by one character.
- If a character in s does not match with either x' or y' , then s is not an interleaving of x' and y' , and return false.

2. **Subproblem Definition, 2D DP Table**
   - We'll create a 2D table opt where opt[i][j] represents whether the substring s[1:i+j] is an interleaving of x'[1:i] and y'[1:j].
   - opt[i][j] would return true if it is the case.
3. **The recurrence Relation:**
   - **If (opt[i-1][j]) = true and s[i+j] = x[i]) or (opt[i][j-1]) = true and s[i+j] = y[j]):**
     opt[i][j] = true

Proof:

Time Complexity:
The algorithm runs in O(n^2) times since we have a 2d table and if we multiply the row by the colum it takes opt[i,j] n^2 to build up. Filling up the table takes constant time.

22. To assess how "well-connected" two nodes in a directed graph are, one can not only look at the length of the shortest path between them, but can also count the *number* of shortest paths.

This turns out to be a problem that can be solved efficiently, subject to some restrictions on the edge costs. Suppose we are given a directed graph $G = (V, E)$, with costs on the edges; the costs may be positive or

negative, but every cycle in the graph has strictly positive cost. We are also given two nodes $v, w \in V$. Give an efficient algorithm that computes the number of shortest v-w paths in G. (The algorithm should not list all the paths; just the number suffices.)

- We would compute the shortest paths to each vertex with length i, and then consider the number of such shortest paths we have.
- Since we don't have a negative cycle and we are trying to find the shortest path, we can basically remove all the positive cycles we have(does not affect the shrotest path), so the total length we are going to iterate between v and w is from 1 to n-1(assuming there are at most n-1 edges between them)
- To compute the shortest path from v to w, there is gonna be a vertex before w (we call it x_j) which is sp(s,x_j), and the total length would be sp(s,x_j) + wx_j,v (wx_j is the weight of the path from x_j to v) So the shortest path between v and w for an arbitrary length i between 1 and n-1 would be:

  sp(v, w) = sp(v, x_j) + w_xj,w

  There are k possible such x_j(x_1 to x_k), and we want to find the minimum of them for the sp of v and w. Therefore, we have:

  For 1<=i<=n-1:

  For 1<=k<=n:

  For 1 <=j<=k:

  sp(v,w) = min (sp(v, x_j)+w_xj, w

Since there are at most n-1 neighbors of w, and there are n possible w values we need to consider(w1, w2, w3 .. since we are considering the shortest path between two arbitrary vertexes in the graph G), therefore we need n^2 computation for every single length i. Since there are i length and length is also at most n, the total time complexity would be o(n^3).

To get the number of min paths, we will create a counter that increments every time we observe a path from v to w with the same wight as our current minimum. If a new optimal path is found, we will simplu reser the counter to 1. This takes constant time, so the time complexity is still 0(n^3).

Proof of Correctness:

- Inductive Basis: The base cases are trivially true by definition of empty string interleaving and direct substring matching for x' and y' with initial segments of s.
- Inductive Step: Assume that opt[i'][j'] is correct for all i' < i and j' < j. Then opt[i][j] relies on the correctness of opt[i-1][j] and opt[i][j-1], both of which have been assumed to be correct. This correctly computes opt[i][j] based on whether s[i+j] can continue the interleaving pattern correctly.

24. *Gerrymandering* is the practice of carving up electoral districts in very careful ways so as to lead to outcomes that favor a particular political party. Recent court challenges to the practice have argued that through this calculated redistricting, large numbers of voters are being effectively (and intentionally) disenfranchised.

Computers, it turns out, have been implicated as the source of some of the "villainy" in the news coverage on this topic: Thanks to powerful software, gerrymandering has changed from an activity carried out by a bunch of people with maps, pencil, and paper into the industrial-strength process that it is today. Why is gerrymandering a computational problem? There are database issues involved in tracking voter demographics down to the level of individual streets and houses; and there are algorithmic issues involved in grouping voters into districts. Let's think a bit about what these latter issues look like.

Suppose we have a set of n *precincts* $P_1, P_2, \ldots, P_n$, each containing m registered voters. We're supposed to divide these precincts into two *districts*, each consisting of $n/2$ of the precincts. Now, for each precinct, we have information on how many voters are registered to each of two political parties. (Suppose, for simplicity, that every voter is registered to one of these two.) We'll say that the set of precincts is *susceptible* to gerrymandering if it is possible to perform the division into two districts in such a way that the same party holds a majority in both districts.

Give an algorithm to determine whether a given set of precincts is susceptible to gerrymandering; the running time of your algorithm should be polynomial in n and m.

Example. Suppose we have $n = 4$ precincts, and the following information on registered voters.

Precinct	**1**	**2**	**3**	**4**
Number registered for party A	55	43	60	47
Number registered for party B	45	57	40	53

This set of precincts is susceptible since, if we grouped precincts 1 and 4 into one district, and precincts 2 and 3 into the other, then party A would have a majority in both districts. (Presumably, the "we" who are doing the grouping here are members of party A.) This example is a quick illustration of the basic unfairness in gerrymandering: Although party A holds only a slim majority in the overall population (205 to 195), it ends up with a majority in not one but both districts.

Imagine there are `n` precincts, each with `m` votes. Define the votes for party A in the `i-th` precinct as `a_i`, and for party B as `b_i`. We use `f_i` to represent the difference `a_i - b_i`.

Our objective is to see if it's possible to split the precincts into two districts in such a way that each district has a majority for a particular party. Initially, we assume that the sum of `f_i` across all precincts is positive, favoring party A. If it's negative, we would simply swap the roles of parties A and B.

The primary question is whether we can divide the precincts into two groups where the sum of `f_i` in each group is positive. This means checking if `f_i` can be partitioned into two groups with positive sums.

We approach this using a dynamic programming technique where `dp[i][p][g][k]` checks if it's feasible to place `k` precincts out of the first `i` to achieve a sum `p` in one district and `g` in the other, with both sums being positive.

Algorithm Steps:

1. Initialize the `dp` array so that `dp[0][p][g][0]` is true for all positive `p` and `g`, indicating no precincts are yet assigned while presuming initial positive sums.
2. Iterate through each precinct, exploring possible sums `p` and `g` for the two districts:
    - For each potential count of assigned precincts `k`, update the `dp` array based on whether adding the current precinct to either district continues to yield positive sums.

The recursive update of the `dp` array relies on the feasibility of previous configurations to maintain positive sums with or without the current precinct in either district.

To determine if a solution exists, examine `dp[n][0][0][n/2]`, assuming an equal division of precincts, to check if such a split is feasible.

Proof of Correctness

This algorithm employs a comprehensive search via dynamic programming, thus ensuring all possible configurations are explored. We prove its accuracy through induction:

- Base Case: Initially, `dp[0][p][g][0]` being true for all positive `p`, `g` means no precincts are yet considered, and the districts are assumed to start with positive values.
- Inductive Step: Assuming correctness up to `i-1` precincts, we extend this to `i` by checking all combinations of `p`, `g`, and `k` for the inclusion of the `i-th` precinct. This maintains the integrity of the distribution according to the previous results.

Time Complexity Analysis

Consider the difference `S = U - L`, where `U` and `L` are the sums of all positive and negative `f_i`, respectively. The four nested loops (over `i`, `p`, `g`, and `k`) each operate within bounds set by `S`. Specifically, lines 3 and 6 run in `O(n^2)`, while lines 4 and 5 scale with `O(S^2)`. Thus, the overall time complexity can be expressed as `O(n^2 * S^2)`, encapsulating the breadth of the algorithm's computational requirements.

7. Consider a set of mobile computing clients in a certain town who each need to be connected to one of several possible *base stations*. We'll suppose there are n clients, with the position of each client specified by its (x, y) coordinates in the plane. There are also k base stations; the position of each of these is specified by (x, y) coordinates as well.

For each client, we wish to connect it to exactly one of the base stations. Our choice of connections is constrained in the following ways.

There is a *range parameter* r—a client can only be connected to a base station that is within distance r. There is also a *load parameter* L—no more than L clients can be connected to any single base station.

Your goal is to design a polynomial-time algorithm for the following problem. Given the positions of a set of clients and a set of base stations, as well as the range and load parameters, decide whether every client can be connected simultaneously to a base station, subject to the range and load conditions in the previous paragraph.

From the problem, we are trying to know if there is a match for each client. (We don't need to prove the max part)

Algorithm:

Design: We will use the max flow to solve this problem.

From the question, we know there are three constraints:
1. Each client connects to exactly one of the base stations.
2. Each client can only be connected to a base station that is within distance r.
3. For each base station within the range r, no more than L clients can be connected to it.

We first put all the clients as the 1st set on the left, and all the base stations as the 2nd set on the right. If the distance between a client(lets say a) and a station is within r, we use an edge between them to represent that. To make sure each client is connected to one base station, we will create a super point called s which points to all the clients. We use flow to represent the connection, so the flow will only flow into one of the edges even if there are tow eges coming out of a cell phone. Also, the edge coming out of s would each has a flow of 1, since again each client should connect to at most 1 station.

For the stations, we will also create a super point that connects T to each of the stations with an edge, each edge has a weight L_j. There can be two different clients connected to the same station, and we use weight and the superstation we created to keep track of the L, so at most L_j clients would match to T_j.

We apply max flow algorithm in this problem since we've already proved that max flow indeed gives us the correct matching between clients and stations.

Proof:
As I've stated above.

Time Complexity:
The time complexity for this algorithm is O(n^2k). The time complexity for Ford-Fullkerson algorithm is O(m|f|), where m is the number of edges and |f| is the max flow. For our case, we have o(n+k) nodes and o(nk) edges at worst. Therefore, we have a runtime complexity of O(n^2k).

Time Complexity of Edmonds-Karp:

The time complexity of the Edmonds-Karp algorithm is $O(VE^2)$, where V is the number of vertices and E is the number of edges in the graph.

- **Vertices**: The graph consists of $n + k$ nodes representing clients and base stations, plus 2 for the source and sink, totaling $V = n + k + 2$.
- **Edges**: There are n edges from the source to each client, k edges from each base station to the sink, and potentially up to $n \times k$ edges between clients and base stations if every client is within range of every station. This results in $E = n + k + n \times k$.

Substituting these into the formula for Edmonds-Karp, the time complexity becomes:
$O((n + k + 2) \times (n + k + n \times k)^2)$
This simplifies approximately to:
$O((n + k) \times (n \times k)^2)$
$O(n^2 \times k^2 \times (n + k))$

9. Network flow issues come up in dealing with natural disasters and other crises, since major unexpected events often require the movement and evacuation of large numbers of people in a short amount of time.

Consider the following scenario. Due to large-scale flooding in a region, paramedics have identified a set of n injured people distributed across the region who need to be rushed to hospitals. There are k hospitals in the region, and each of the n people needs to be brought to a hospital that is within a half-hour's driving time of their current location (so different people will have different options for hospitals, depending on where they are right now).

At the same time, one doesn't want to overload any one of the hospitals by sending too many patients its way. The paramedics are in touch by cell phone, and they want to collectively work out whether they can choose a hospital for each of the injured people in such a way that the load on the hospitals is *balanced*: Each hospital receives at most $\lceil n/k \rceil$ people.

Give a polynomial-time algorithm that takes the given information about the people's locations and determines whether this is possible.

- Graph Design:
  - Create a source super node S
  - Create a super sink node t
  - Add nodes for each injured person and each hospital.
  - Connect s to each person with an edge of capacity 1(each person needs exactly one hospital)
  - Connect each hospital to T with edges of capacity with edges of capacity [n/k](the maximum number of patients per hospital)
  - Connect persons to hospital based on the distance constraint(only if within a half-hour's drive)

Proof:

Modeling as Network Flow:

1. Graph Representation: We model the problem as a flow network where:

- Each injured person is a node connected to a source node s with an edge of capacity 1. This represents that each person must be assigned to exactly one hospital.
- Each hospital is a node connected to a sink node t with an edge of capacity[n/k]. This ensures that no hospital receives more than this number of patients, maintaining balance.
- An edge from a person to a hospital node exists if the hospital is within a half-hour's drive, indicating potential assignment.

Flow Properties:

2. Integral Flows: The capacities in this network are integers (1 and $\lceil n/k \rceil$), so the max-flow min-cut theorem ensures that there exists an integral maximum flow. This means we can find a flow where the amount of flow through the network (representing the assignment of patients to hospitals) is entirely composed of whole numbers, suitable for our requirement that each person is fully assigned to one hospital.

3. Maximum Flow Equals Number of Injured People: For the solution to be valid, the maximum flow from S to T must equal n, the total number of injured people. This condition ensures that every injured person is assigned to a hospital within the required constraints.
4. Feasibility of Assignments: If the max-flow algorithm finds a flow of value n, it means it's possible to route one unit of flow from S to T for each person, respecting the capacity constraints imposed by hospital capacities and the connectivity constraints imposed by the driving distance. No more patients than the hospital capacity are assigned, and all assignments respect the geographical constraints.
5. Optimality and Exhaustiveness: By the construction of the flow network and the properties of the max-flow algorithm, if a valid assignment configuration exists, the algorithm will find it. If it's not possible to assign every patient while respecting the constraints, the maximum flow will be less than n, correctly indicating that no valid assignment exists.

Time Complexity:

The time complexity for this algorithm is O(n^2k). The time complexity for the Ford-Fullkerson algorithm is O(m|f|), where m is the number of edges and |f| is the max flow. For our case, we have o(n+k) nodes and o(nk) edges at worst. Therefore, we have a runtime complexity of O(n^2k).

Problem 6. A heroic helicopter named Chopper is on a mission to rescue injured hikers in a mountain range. Due to an old mechanical quirk, Chopper can only maintain a consistent rhythm of ascending and descending. Each time it climbs to a higher peak, its next trip must descend to a lower one before climbing again, and vice versa.

Chopper has received several distress calls from hikers stranded on various mountain peaks. Chopper's goal is to rescue as many hikers as possible by finding the longest subsequence of mountains that satisfy the requirement of alternating between ascending and descending heights. Given the heights of the mountains where the hikers are located, suggest an algorithm to guide Chopper.

Example:

- Heights: 8, 9, 6, 4, 5, 7, 3, 2, 4
- Output**:** 8, 9, 6, 7, 3, 4

Explanation: 8 < 9 > 6 < 7 > 3 < 4 (length: 6). Any other subsequence of mountains with the same length (6) is acceptable.

a[i]: The i-th element of the input array.

dp_s[i]: Length of the longest alternating subsequence ending at index i where the last element is smaller than the previous element.

dp_l[i]: Length of the longest alternating subsequence ending at index i where the last element is greater than the previous element.

res: The length of the longest alternating subsequence found across the entire array.

Initially, every position in dp_s and dp_l is set to 1 because any single element is a subsequence of length 1 by itself.

Algorithm Steps:

Initialize dp_s and dp_l Arrays:

- Each position is initialized to 1 since any individual element is a trivial alternating subsequence.

Iterate Over the Array:

- For each element a[i] from the first to the last:

Check All Previous Elements:

- For each previous element a[k] where k < i:
    - If a[k] is greater than or equal to a[i], update dp_s[i]. This checks if you can form a longer subsequence ending in a[i] that decreases at the end. Update dp_s[i] to be the maximum of its current value or dp_l[k] + 1 (indicating you are adding a[i] to a subsequence that last increased at a[k]).
    - If a[k] is less than a[i], update dp_l[i]. This checks if you can form a longer subsequence ending in a[i] that increases at the end. Update dp_l[i] to be the maximum of its current value or dp_s[k] + 1 (indicating you are adding a[i] to a subsequence that last decreased at a[k]).

Track the Maximum Subsequence Length:

- After processing each a[i], update res to hold the maximum value between the current res, dp_s[i], and dp_l[i]. This ensures that res always contains the length of the longest alternating subsequence found so far.

Return the Result:

- After all elements have been processed, return res, which now holds the length of the longest alternating subsequence in the array.

Time Complexity:

The time complexity of this algorithm is O(n^2), where n is the number of elements in the array. This is because for each element i, the algorithm potentially compares it against all previous elements k, resulting in a quadratic number of comparisons.