# DYNAMIC PROGRAMMING

# Problem Statement

We are given an array Arr[] of length n. It represents the price of a stock on ‘n’ days. The following guidelines need to be followed:

1. We can buy and sell the stock any number of times.
2. In order to sell the stock, we need to first buy it on the same or any previous day.
3. We can’t buy a stock again after buying it once. In other words, we first buy a stock and then sell it. After selling we can buy and sell again. But we can’t sell before buying and can’t buy before selling any previously bought stock.

**Allowed**

B S B S
Arr: [7, 1, 5, 3, 6, 4]

---

**Not Allowed**

S B
Arr: [7, 1, 5, 3, 6, 4]

**Not Allowed**

B B
Arr: [7, 1, 5, 3, 6, 4]

**Not Allowed**

B S S
Arr: [7, 1, 5, 3, 6, 4]

Subproblem Definition?

# Subproblem definition

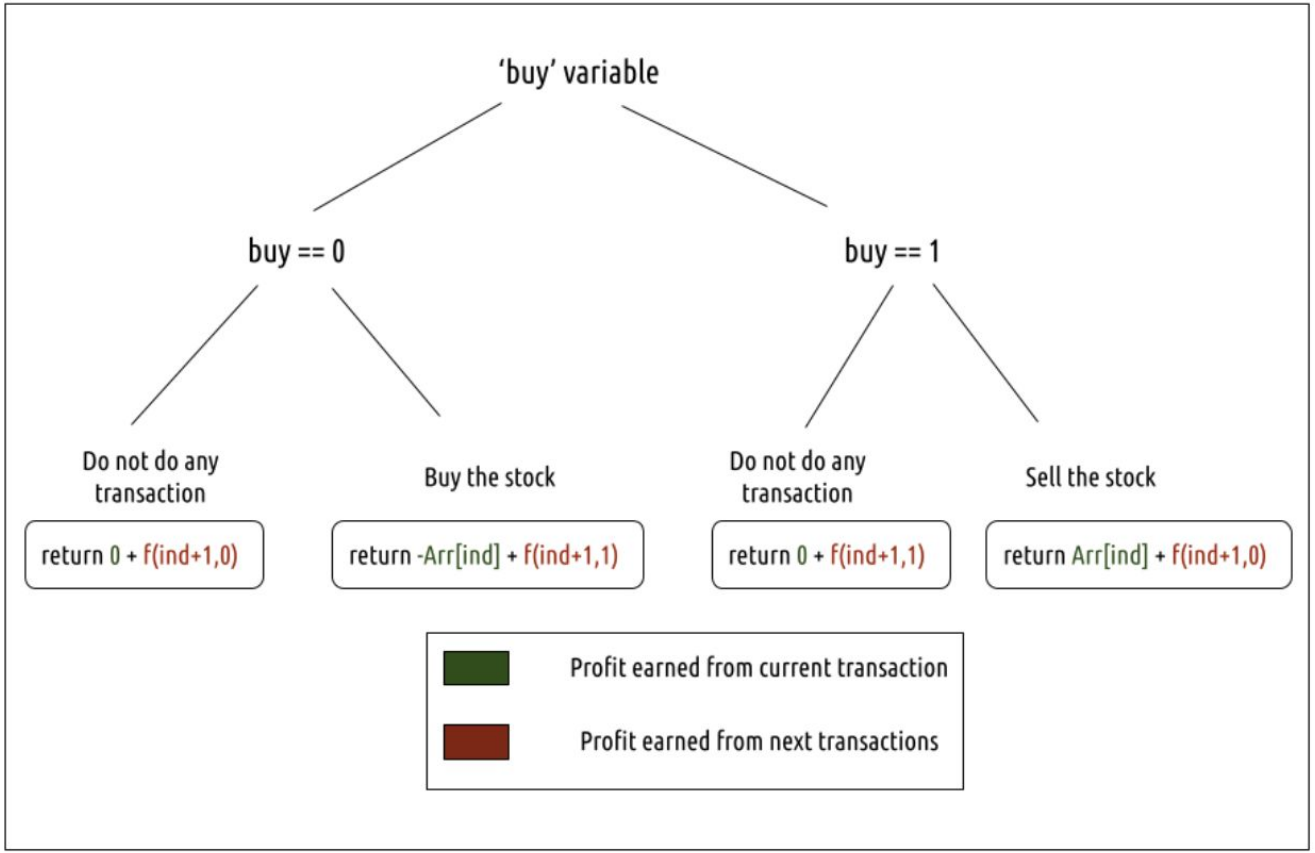
f(ind,buy) -> The maximum profit we can generate from day ind to day n-1, where 'buy' tell that we can buy/sell the stock at day ind.

Ind -> Array index

buy -> value 0/1 [ 0 -> we can buy the stock on that day]

[ 1 -> we can't buy the stock on that day, we can sell the stock]

# Why dynamic programming ? - overlapping subproblems



Recurrence Relation(s)?

# Recurrence relations

```
f(ind,buy) {

      if(buy==0){
        op1 = 0 + f(ind+1,0) // do no transaction

        op2 = -Arr[ind] + f(ind+1,1)  // buy  the stock
      }

      if(buy==1){
        op1 = 0 + f(ind+1,1) // do no transaction

        op2 = Arr[ind] + f(ind+1,0)  // sell  the stock
      }
      return  max(op1,op2)
}
```

Base case - If ind==n, it means we have finished trading on all days, and there is no more money that we can get, therefore we simply return 0.

# Pseudo-code

```
dp[n][0] = dp[n][1] = 0      // Base condition: If we have no stocks to buy or sell, profit is 0


profit = 0      // Iterate through the array in reverse to calculate the maximum profit
for ind from n - 1 to 0:
        for buy from 0 to 1:
                if buy == 0:  // We can buy the stock
                        profit = max(0 + dp[ind + 1][0], -Arr[ind] + dp[ind + 1][1])
                else if buy == 1:  // We can sell the stock
                        profit = max(0 + dp[ind + 1][1], Arr[ind] + dp[ind + 1][0])
                dp[ind][buy] = profit
return dp[0][0]  // The maximum profit is stored at dp[0][0]
```

# Complexities

Time Complexity - O(2*n)

Space Complexity - O(2*n) (can it be reduced?)

# Problem Statement

We are given an array Arr[] of length n. It represents the price of a stock on ‘n’ days. The following guidelines need to be followed:

1. In order to sell the stock, we need to first buy it on the same or any previous day.
2. We can’t buy a stock again after buying it once. In other words, we first buy a stock and then sell it. After selling we can buy and sell again. But we can’t sell before buying and can’t buy before selling any previously bought stock.
3. **We can do at most 2 transactions.**

Subproblem Definition?

# Subproblem definition

f(ind,buy,cap) -> The maximum profit we can generate from day 'ind' to day 'n-1', where 'buy' tell that we can buy/sell the stock at day 'ind' and 'cap' tells us the number of transactions left.

Ind -> Array index

buy -> value 0/1 [ 0 -> we can buy the stock on that day]

[ 1 -> we can't buy the stock on that day, we can sell the stock]

cap -> Number of transactions left

Recurrence Relation(s)?

# Recurrence relations and base case

```
f(ind,buy,cap) {
        If (ind == n || cap == 0) return 0

        if(buy==0){
           op1 = 0 + f(ind+1,0,cap) // do no transaction

           op2 = -Arr[ind] + f(ind+1,1,cap)  // buy  the stock
        }

        if(buy==1){
           op1 = 0 + f(ind+1,1,cap) // do no transaction

           op2 = Arr[ind] + f(ind+1,0,cap-1)  // sell  the stock
        }
        return max(op1,op2)
    }
```

# Pseudo-code

```
dp[ind][buy][cap] = 0 wherever cap == 0 or ind == n

for ind from n - 1 to 0:
        for buy from 0 to 1:
                for cap from 1 to 2:
                        if buy == 0:  // We can buy the stock
                                dp[ind][buy][cap] = max(0 + dp[ind + 1][0][cap], -prices[ind] + dp[ind + 1][1][cap])
                        else if buy == 1:  // We can sell the stock
                                dp[ind][buy][cap] = max(0 + dp[ind + 1][1][cap],
                        prices[ind] + dp[ind + 1][0][cap - 1])

// The maximum profit with 2 transactions is stored in dp[0][0][2]
return dp[0][0][2]
```

# Complexities

Time Complexity - O(n*2*3)

Space Complexity - O(n*2*3) (can it be reduced again?)

# Problem Statement

We are given an array Arr[] of length n. It represents the price of a stock on ‘n’ days. The following guidelines need to be followed:

1. We can buy and sell the stock any number of times.
2. In order to sell the stock, we need to first buy it on the same or any previous day.
3. We can’t buy a stock again after buying it once. In other words, we first buy a stock and then sell it. After selling we can buy and sell again. But we can’t sell before buying and can’t buy before selling any previously bought stock.
4. **We can’t buy a stock on the very next day of selling it. This is the cooldown clause.**

Subproblem Definition?

# Subproblem definition

f(ind,buy) -> The maximum profit we can generate from day ind to day n-1, where 'buy' tell that we can buy/sell the stock at day ind.

Ind -> Array index

buy -> value 0/1 [ 0 -> we can buy the stock on that day]

[ 1 -> we can't buy the stock on that day, we can sell the stock]

Recurrence Relation(s)?

# Recurrence relations and base cases

```
f(ind,buy) {

      if(buy==0){
         op1 = 0 + f(ind+1,0) // do no transaction

         op2 = -Arr[ind] + f(ind+1,1)  // buy  the stock
      }

      if(buy==1){
         op1 = 0 + f(ind+1,1) // do no transaction

         op2 = Arr[ind] + f(ind+2,0)  // sell  the stock
      }
       return  max(op1,op2)
}
```

Base case - If ind>=n, it means we have finished trading on all days, and there is no more money that we can get, therefore we simply return 0.

# pseudo-code

```
dp[ind][buy] = 0 wherever ind>=n

for ind from n - 1 to 0:
        for buy from 0 to 1:
                profit = 0
                if buy == 0:  // We can buy the stock
                        profit = max(0 + dp[ind + 1][0], -Arr[ind] + dp[ind + 1][1])
                else if buy == 1:  // We can sell the stock
                        profit = max(0 + dp[ind + 1][1], Arr[ind] + dp[ind + 2][0])
                dp[ind][buy] = profit

// The maximum profit is stored in dp[0][0]
return dp[0][0]
```

# Complexities

Time complexity - O(2*n)

Space complexity - O(2*n) (can it be reduced?)

# Question 1

You are asked by the city mayor to organize a COVID-19 panel discussion regarding the reopening of your town. He told you that panel members include two types of people: those who wear a face covering (F) and those who do not wear any protection (P). He also told you that to reduce the spread of the virus, those who do not wear any protection must not be sitting next to each other in the panel. Suppose that there is a row of n empty seats. The city mayor wants to know the number of valid seating arrangements for the panel members you can do.

To help you see the problem better, suppose you have n=3 seats for the panel members.

- Some valid seating arrangements you can do are: F-F-F, F-P-F, P-F-P.
- Some invalid seating arrangements are: F-**P-P**, **P-P-P**.

Describe a dynamic programming solution to solve this problem.

# Question 1

a). Define (in plain English) subproblems to be solved.

Solution: Let f(n) be the number of valid seating arrangement for the panel members when you have n seats.

# Question 1

b). Write a recurrence relation for f(n). Be sure to state base cases

Solution:

- First, we can take for each valid seating of n-1 members and put F at the n-th seat.
- What about P at the n-th seat? We need to take a look at valid seating arrangements of n-2 members. Here, we can take for each valid seating of n-2 members, put F at the (n-1)-th seat and then put P at the n-th seat.
- This exhausts all the ways of getting a valid seating. Hence, the recurrence is $f(n) = f(n-1) + f(n-2)$.

# Question 1

c). Use the recurrence part from a and write pseudocode

Solution:

Initialize array f.

Set base cases f[0] = 0, f[1] = 2, f[2]= 3.

For i>2 to n :

    f[i] = f[i-1] + f[i-2]

return f[n]

# Question 1

d). What is the run time of the algorithm?

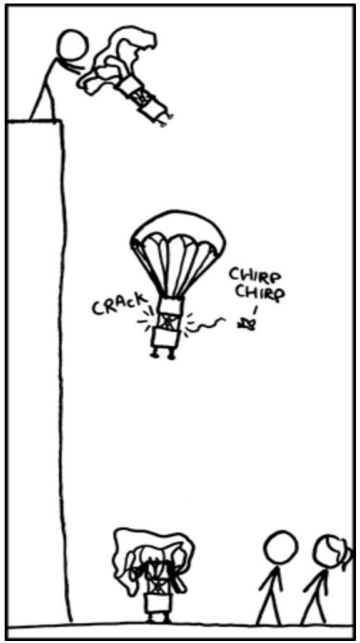
Solution: Looking up the pre-computed value takes O(1) and we only need to store n unique sub-problem we encounter. Hence, this will take O(n) time.

# Question 2

Suppose you are in an n-story building and you have k eggs in your bag. You want to find out the lowest floor from which dropping an egg will break it. What is the minimum number of egg-dropping trials that is sufficient for finding out the answer in all possible cases?

# Question 2: Example

- *For example, if the building has 100 floors (n = 100)*
  - *we have only 1 egg, then we must do 100 trials to find the threshold*
  - *we have 2 eggs, then we can find the threshold in 14 trials!*




XKCD 510, alt text: “I hear my brother Ricky won his school’s egg drop by leaving the egg inside the hen.”

# Question 2: Constraints

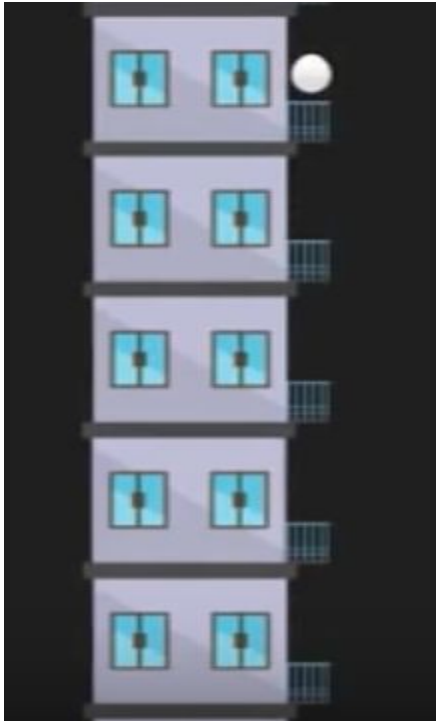
Following are the constraints:

- An egg that survives a fall can be used again.
- A broken egg must be discarded.
- The effect of a fall is the same for all eggs.
- If an egg breaks from a fall, then it would break if dropped from a higher floor.
- If an egg survives a fall then it would survive a shorter fall.

Remember, We are finding the least number of drops to find the threshold floor and not the threshold floor itself!

# Question 2: Base Cases

- If we there are n floors and 1 egg, we need to do n trials worst case
- If there are k eggs and 0 floors, we would need to do 0 trials
- If there are k eggs and 1 floor, we need to do 1 trial



# Define the subproblem

- Let OPT[i, j] denote the minimum number of trials required for an i story building and j eggs

# Find the Recurrence for OPT[i, j]

- Obtain OPT[i, j] given all the results of the solved subproblems
- Suppose we first drop an egg from floor x <= i.
  - If the egg breaks, then we learn that threshold <= x, but we are left with j - 1 eggs. To find out the threshold from this state we would need extra OPT[x – 1, j - 1] steps.
  - If the egg doesn't break, then we learn that x < threshold and we still have k eggs. All the above (i - x) floors are the possible candidates and we have j eggs. Therefore, we would need to do OPT[i – x, j] extra steps.
- We must traverse for each floor x from 1 to i to find the minimum
- $OPT[i, j] = \min_{1<=x<=i}\{1 + \max(OPT[x – 1, j - 1], OPT[i – x, j])\}$

# PseudoCode

- Base Case:
  - OPT[0, j] = 0; // 0 floor
  - OPT[1, j] = 1; // 1 floor
  - OPT[i, 1] = i;  // 1 egg
  - OPT[i, j] = MAX_INTEGER, otherwise
- Iteration:

```
for j = 2 to k:
    for i = 2 to n:
        for x = 1 to i:
            OPT[i, j] = min(OPT[i, j], 1 + max(OPT[x – 1, j - 1], OPT[i – x, j]))
```

- Return OPT[n, k]

# Complexity Analysis

- Time complexity is $O(n^2k)$
- Space complexity is $O(nk)$

# Question 3

You are given an integer array nums. Two players are playing a game with this array: player 1 and player 2.

Player 1 and player 2 take turns, with player 1 starting first. Both players start the game with a score of 0. At each turn, the player takes one of the numbers from either end of the array (i.e., nums[0] or nums[nums.length - 1]) which reduces the size of the array by 1. The player adds the chosen number to their score. The game ends when there are no more elements in the array.

Return true if Player 1 can win the game. If the scores of both players are equal, then player 1 is still the winner, and you should also return true. You may assume that both players are playing optimally.

# Question 3

**Example**

**Input:** nums = [1,5,2]

**Output:** false

**Explanation:**

- Initially, player 1 can choose between 1 and 2.
- If he chooses 2 (or 1), then player 2 can choose from 1 (or 2) and 5. If player 2 chooses 5, then player 1 will be left with 1 (or 2).
- So, final score of player 1 is 1 + 2 = 3, and player 2 is 5.
- Hence, player 1 will never be the winner and you need to return false.

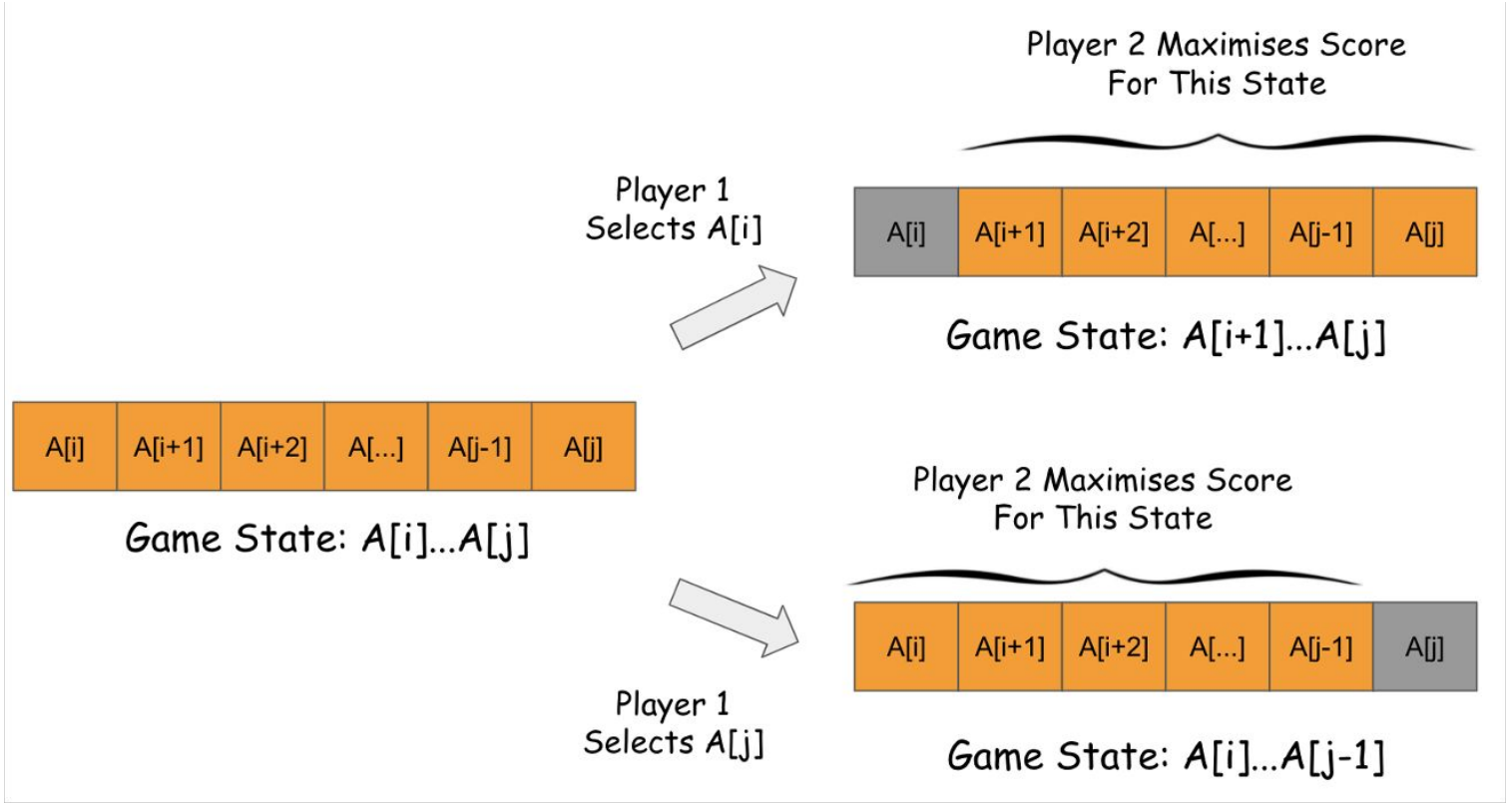
# Question 3

Knowing that this is an optimization problem (score maximization), should be a hint that we need a greedy or DP algorithm. To formulate a DP algorithm, we must think about what the sub-problems should be and how they can be recursively computed.

From experience, the **sub-problems for arrays are subsets of the array.**

The recursive relationship can be figured out by thinking about the choices we make to construct our solution. In this case, we are picking numbers, so the choice could be **'Which number was picked — the one at the start or the one at the end?'.**

# Question 3


Player 2 Maximises Score
For This State
Player 1
Selects A[i]
A[i]
A[i+1]
A[i+2]
A[...]
A[j-1]
A[j]
Game State: A[i+1]...A[j]
A[i]
A[i+1]
A[i+2]
A[...]
A[j-1]
A[j]
Game State: A[i]...A[j]
Player 2 Maximises Score
For This State
A[i]
A[i+1]
A[i+2]
A[...]
A[j-1]
A[j]
Player 1
Selects A[j]
Game State: A[i]...A[j-1]

# Question 3

Let dp[i][j] store the maximum effective score won by player 1 for the subarray nums[i:j]

**Choice 1:**
If Player 1 picks nums[i], nums[i] is added to the score and the game state nums[(i+1)...j] is given to Player 2. We know that Player 2 plays optimally as well so it will maximize its score for the game state presented which is dp[i+1][ j]. This leaves the remaining numbers for Player 1: (nums[i+1] + nums[i+2] + … + nums[j]) - dp[i+1][j].

**Choice 2:**
If Player 1 picks nums[j], nums[j] is added to the score and the game state nums[(i...(j-1)] is given to Player 2. We know that Player 2 plays optimally as well so it will maximize its score for the game state presented which is dp[i][j-1]. This leaves the remaining numbers for Player 1:  (nums[i] + nums[i+1] + … + nums[j-1]) - dp[i][j-1].

# Question 3

Let cumSum be a 1D array such that cumSum[i] = nums[0] + nums[1] + ... + nums[i].

Player 1 picks nums[i]:
- Player 2 gains dp[i+1][j]
- The remaining score for Player 1 is (cumSum[j] - cumSum[i+1] + nums[i+1]) - dp[i+1][j]

v1 = nums[i] + (cumSum[j] - cumSum[i+1] + nums[i+1]) - dp[i+1][j]

Player 1 picks nums[j]:
- Player 2 gains dp[i][j-1]
- The remaining score for Player 1 is (cumSum[j-1] - cumSum[i] + nums[i]) - dp[i][j-1]

v2 = nums[j] + (cumSum[j-1] - cumSum[i] + nums[i]) - dp[i][j-1]

Thus, dp[i][j] = max{v1, v2}

# Question 3

To summarise the discussion above:

- Define dp[n][n] such that dp[i][j] (j > i) represents the maximum effective scores won by the first player playing with the game state nums[i...j].
- Define cumSum[]such that cumSum[i] = nums[0] + nums[1] + … + nums[i].
- (Base Case) dp[i][i] = nums[i] and dp[i][i+1] = max(nums[i], nums[i+1]) because the first player will pick the bigger number.
- v1 = nums[i] + cumSum[j] - cumSum[i+1] + nums[i+1]- dp[i+1][j]

  v2 = nums[j] + cumSum[j-1] - cumSum[i] + nums[i]- dp[i][j-1]

  dp[i][j] = max(v1, v2)
- Final answer is dp[0][n-1] ≥ cumSum[n-1] - dp[0][n-1].

# Question 3

**Complexity Analysis**

**Time Complexity**

Computing the cumulative sums array takes O(n) time. To compute each entry of DP table, we are looking at two possible choices that Player 1 can make. For each choice, computing the maximum score takes O(1) time. Thus, computing each entry of dp[][] takes O(1) time. The overall runtime is $O(n^2) \times O(1) = O(n^2)$.

**Space Complexity**

Our 2D DP table has $O(n^2)$ entries, $O(n^2)$ in total

# Network Flow

# True/False

# True/False

**(T/F)** If $\sum_v d_v \neq 0$, then there is no feasible circulation in the graph for the set of demands $\{d_v\}$.

True!

# True/False

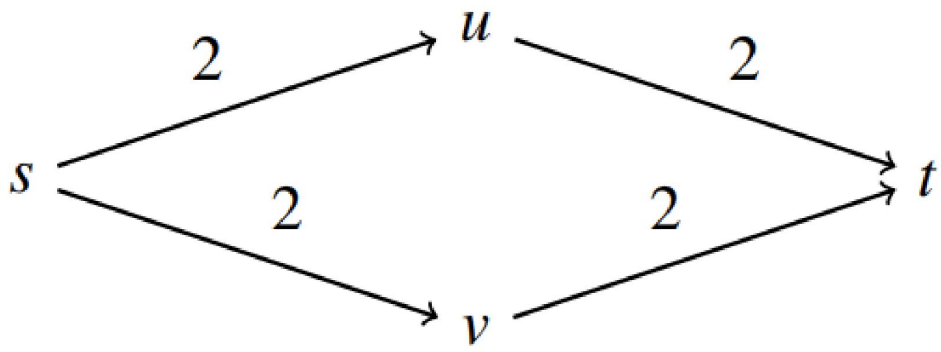
**(T/F)** Given a graph $G$ and a total demand value of $D$ if the absolute value of the max flow $|f|$ is less than $D$ then a feasible circulation exists in $G$.

False.

# True/False

**(T/F)** Suppose the maximum (s,t)-flow of some graph has value $f$. Now we increase the capacity of every edge by 1. Then the maximum (s,t)-flow in this modified graph will have value at most $f+1$.

False. Consider the counterexample.



# Questions

# Network Flow

You are employed at a video game company, which is preparing to release $n$ distinct types of video games. Before their release, these games need to be tested. There are $m$ players available for testing, where $m \geq n$. Each game type appeals to a specific subset of these players. The goal is to distribute these $n$ game types among the players for testing, adhering to the following conditions:

a) Each game type should only be given to players who have shown interest in that specific type.

b) A player can test a maximum of $k$ different game types.

c) No player should receive the same game type more than once.

d) The objective is to maximize the total number of game types distributed for testing.

Design an efficient network flow-based algorithm for this problem. You are required to make a claim regarding the correctness of your algorithm and provide proofs in both directions to support your claim.

# Step 1: Nodes

Look into the **objects** in the problem statement.

You are employed at a video game company, which is preparing to release $n$ distinct types of video games. Before their release, these games need to be tested. There are $m$ players available for testing, where $m \geq n$. Each game type appeals to a specific subset of these players. The goal is to distribute these $n$ game types among the players for testing, adhering to the following conditions:

Two kinds of nodes: game types, and players.

The players need to “test” the game types. The relationship “test” can be regarded as one kind of edges. Importantly, one test of the player to the game type is regarded as **one unit of flow**.

# Step 2: Constraints

Look into the related constraints in the problem statement.

Only the players in P[i] can test game type i

A player can test at most K game types

A player at most tests a game type once

You are employed at a video game company, which is preparing to release $n$ distinct types of video games. Before their release, these games need to be tested. There are $m$ players available for testing, where $m \geq n$. Each game type appeals to a specific subset of these players. The goal is to distribute these $n$ game types among the players for testing, adhering to the following conditions:

a) Each game type should only be given to players who have shown interest in that specific type.

b) A player can test a maximum of $k$ different game types.

c) No player should receive the same game type more than once.

# Step 3: Convert Constraints

Constraint 1: only the **players** in P[i] can test **game type** i

=> create edges from game type i to the players in P[i]

Constraint 2: a **player** can test at most **K** times

=> create edges from player to sink

set the capacity of these edges to **K**

Usually, if the constraint is **only** related to one type of nodes, the converted edges would be linked with **source** of **sink** node.

Constraint 3: a **player** at most tests a **game type** once

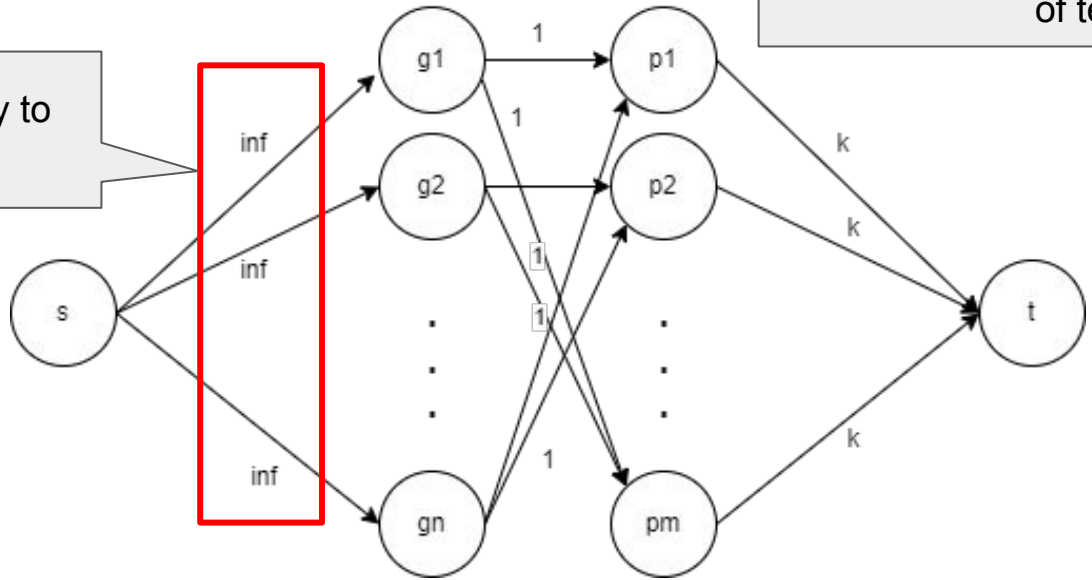
=> set the capacity for edges between game types and players to **1.**

# Step 4: Objective, and Other Edges

d) The objective is to maximize the total number of game types distributed for testing.

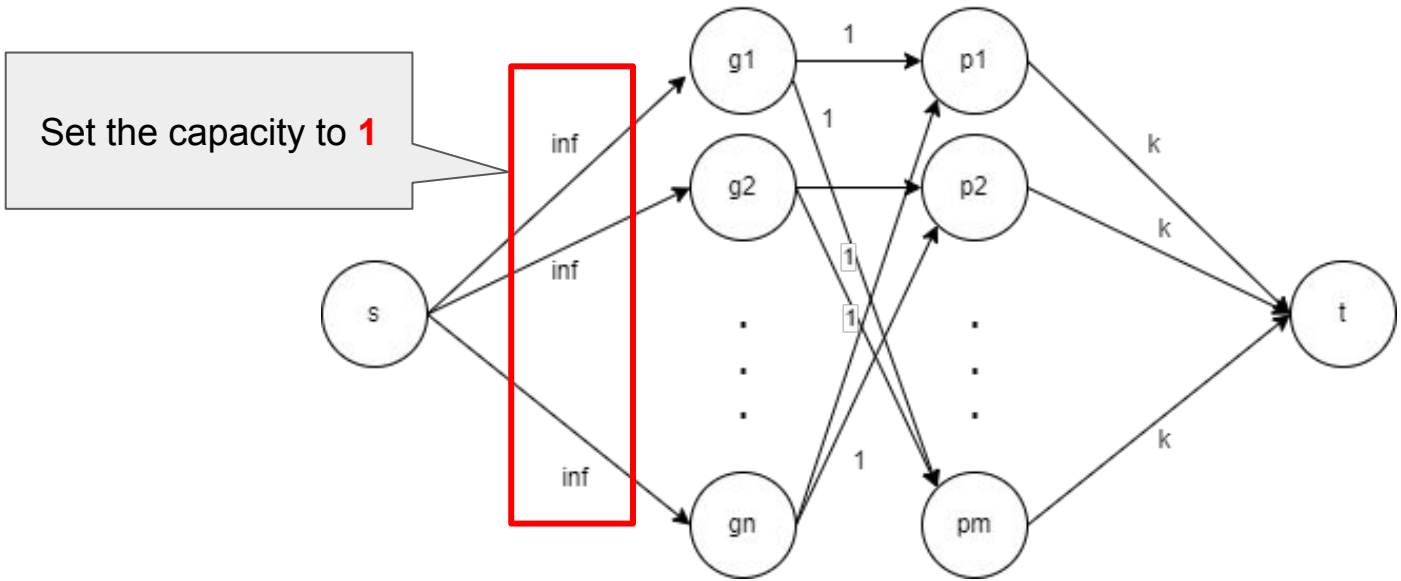
In other words, maximize the number of tests

Set the capacity to infinity



# Follow Up Question

If the objective is to maximize **the number of tested game types**, what modifications do we need to make?



# Proof

Don’t forget to prove the correctness of your designed algorithm!

Design an efficient network flow-based algorithm for this problem. You are required to make a claim regarding the correctness of your algorithm and provide proofs in both directions to support your claim.

The claim is **required**! Usually, “the desired answer of this question is equal to the maxflow (or sth else) of the constructed graph G.

Proofs in **both** directions.
You need to show that:
(1) If desired answer of this question is A, then the maxflow equals to A;
(2) if the maxflow is A, then the desired answer is A.

# Network Flow Solution

- **Claim**: We can maximize the total number of video games given if and only if we maximize the flow in graph G.
- **Forward**: Any flow translates to a valid solution (satisfying all the constraints)
- **Backward**: Any valid solution to the distribution problem can be mapped to a valid flow for the graph.
- Flow out of s is the total number of games distributed. Therefore, we maximize the total number of games distributed for testing.

# Circulation

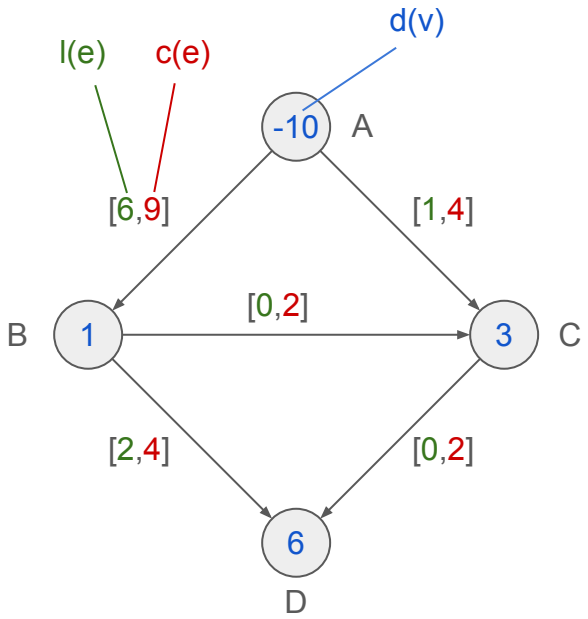
Review Session - Exam 2
CSCI 570 Spring 2024

# Circulation with Demands and Lower Bounds

Given a directed graph $G = (V, E)$, demands $d(v)$ for each vertex $v$, and lower bounds $l(e)$ and capacities $c(e)$ for each edge $e$, the circulation problem asks you to find a flow $f$ such that:

1) $f^{in}(v) - f^{out}(v) = d(v)$ for all $v \in V$

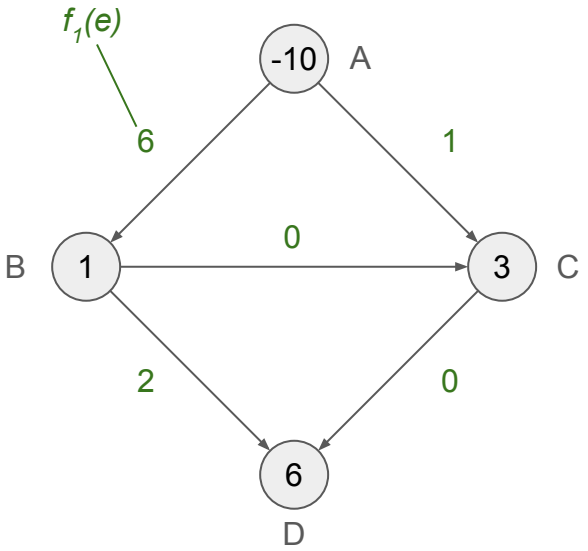
2) $l(e) \leq f(e) \leq c(e)$ for all $e \in E$



# Finding Circulation

Solution

1. Set flow $f_1(e) = l(e)$ for all edges $e$



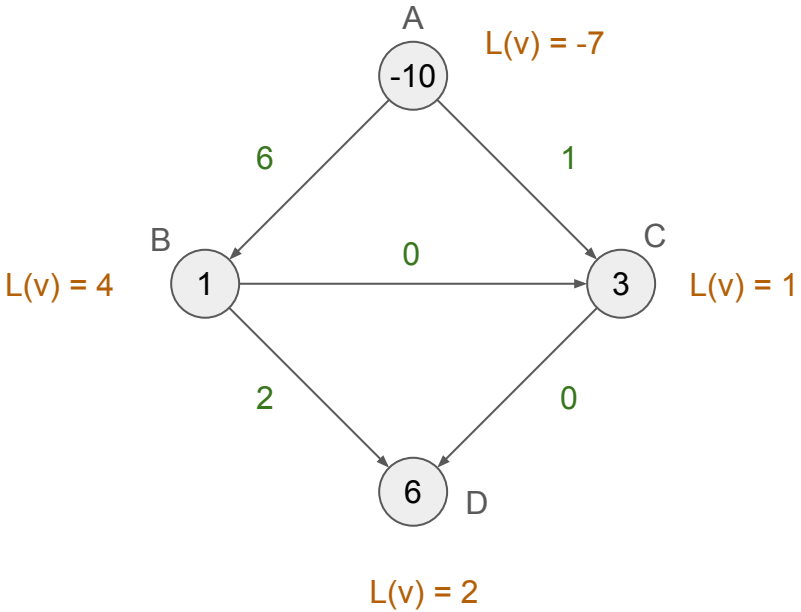
# Finding Circulation

Solution

1. Set flow $f_1(e) = l(e)$ for all edges $e$

2. Calculate the imbalance $L(v)$ at each vertex $v$

$$L(v) = f_1^{in}(v) - f_1^{out}(v)$$

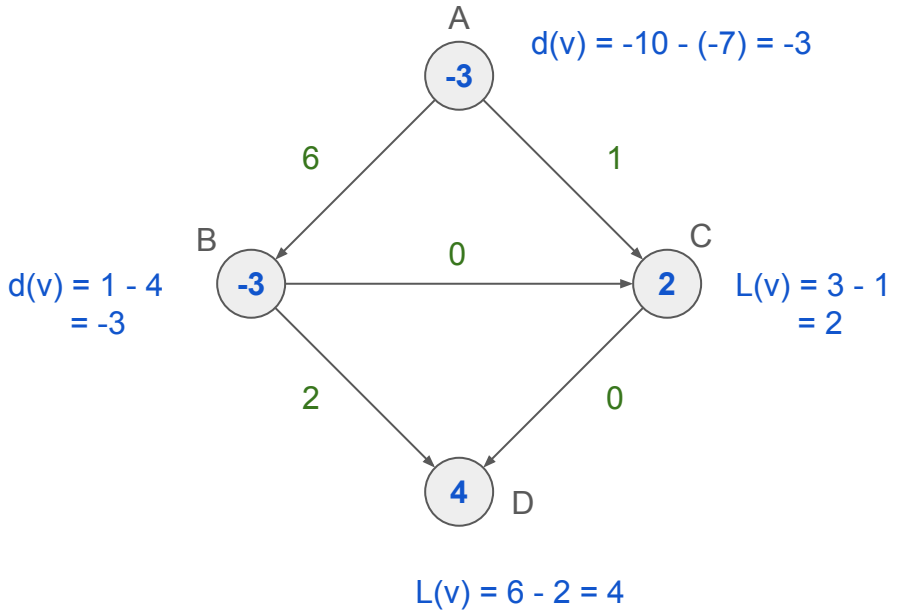


# Finding Circulation

Solution

3. Set new demand $d^{new}(v)$ for each vertex $v$

$$d^{new}(v) = d^{old}(v) - L(v)$$



# Finding Circulation

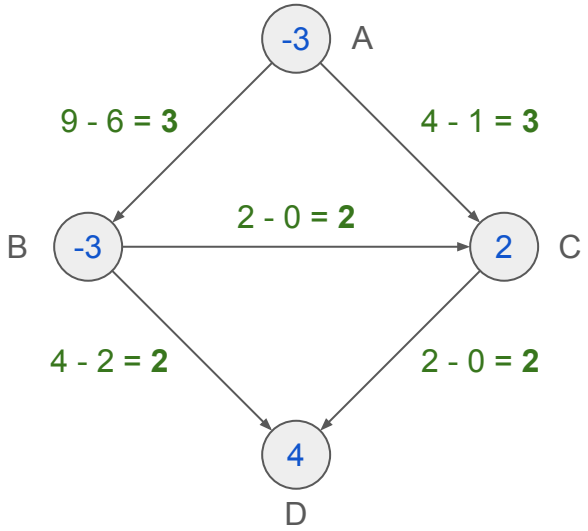
Solution

3. Set new demand $d^{new}(v)$ for each vertex $v$

$$d^{new}(v) = d^{old}(v) - L(v)$$

4. Set new capacity $c^{new}(e)$ for each edge $e$

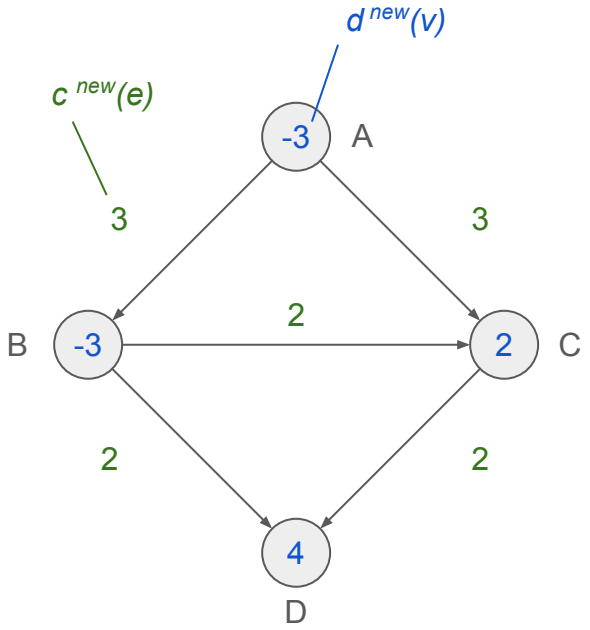
$$c^{new}(e) = c^{old}(e) - l(e)$$



# Finding Circulation

Solution

We have converted the problem of finding circulation with demands + lower bounds to finding circulation with demands only

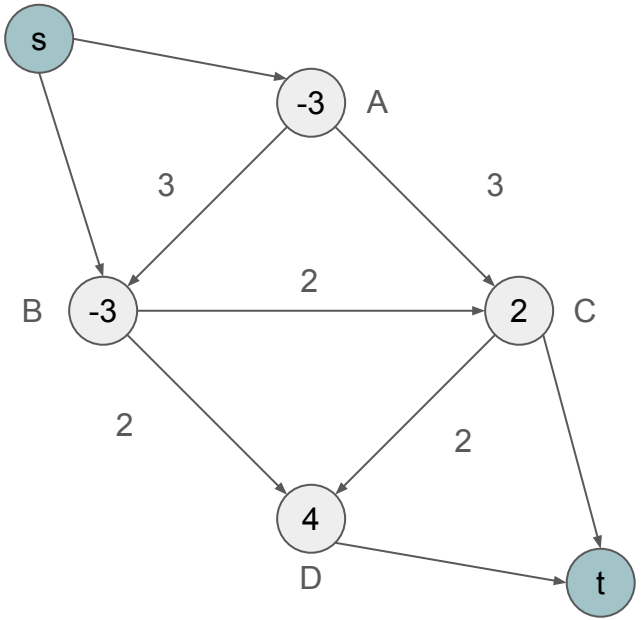


# Finding Circulation

Solution

5. Connect vertices with negative demands to supersource *s*, and vertices with positive demands with supersink *t*

   $V = V \cup \{s, t\}$
   $E = E \cup \{ (s, u) \mid d(u) < 0 \}$
   $\cup \{ (v, t) \mid d(v) > 0 \}$

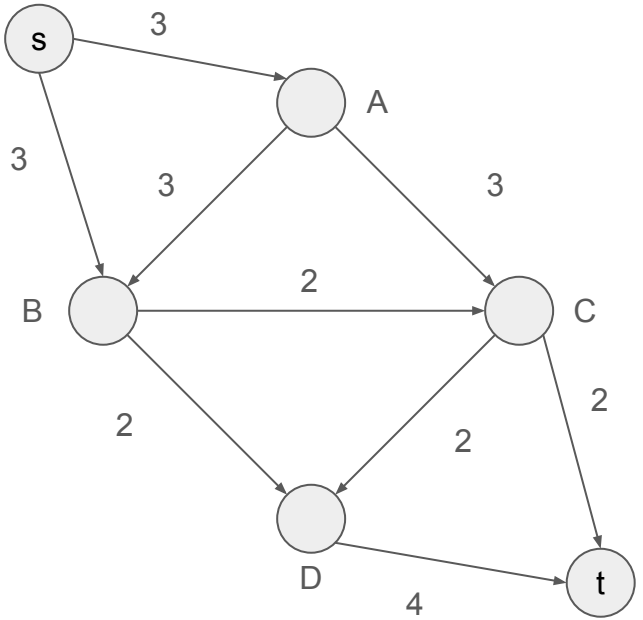


# Finding Circulation

Solution

6. Set $c(s, u) = -d(u)$ and $c(v, t) = d(v)$

   Set $d(v) = 0$ for all vertex $v \in V - \{s, t\}$



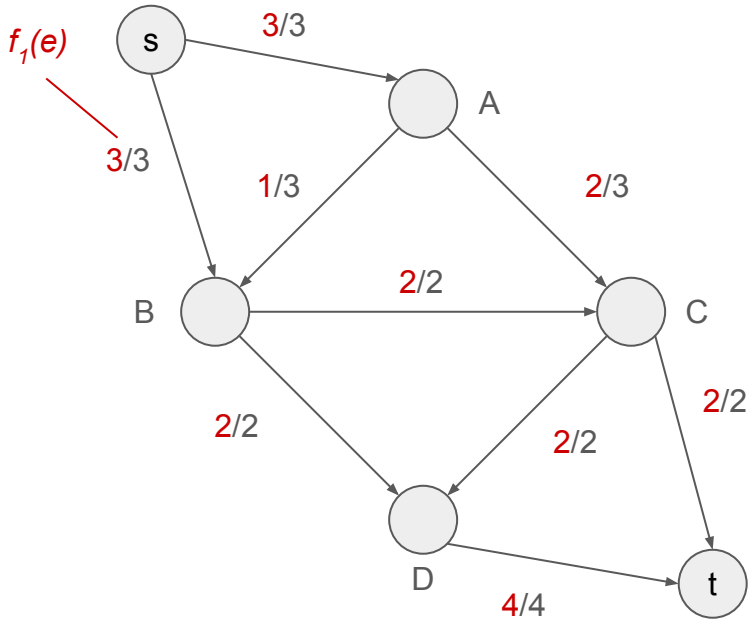
# Finding Circulation

Solution

6. Set $c(s, u) = -d(u)$ and $c(v, t) = d(v)$

   Set d(v) = 0 for all vertex $v \in V - \{s, t\}$

7. Find maximum flow $f_2$ in the new network

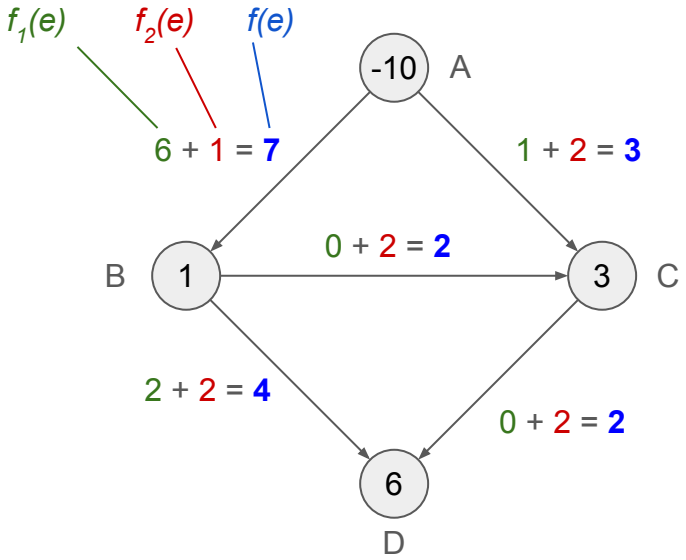


# Finding Circulation

Solution

8. Set $f = f_1 + f_2$
   Remove *s* and *t*

   The flow *f* is the solution



# Problem

For Fall & Spring 2024, ***n*** students want to graduate. There are ***m*** total courses. Following conditions should be met across both semesters:

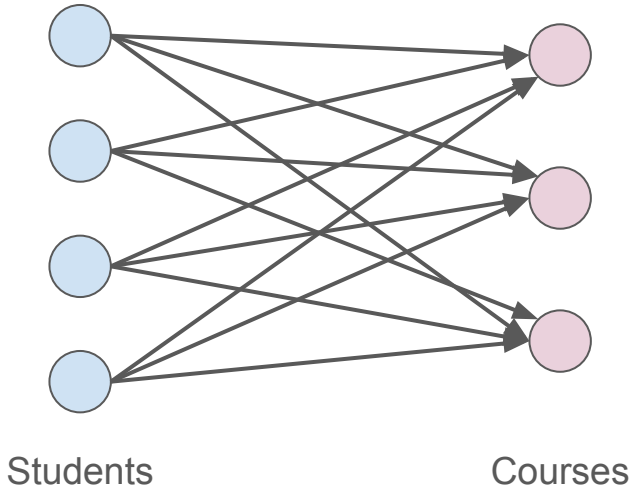
1. Student ***i*** must take $r_i$ number of courses.
2. Student ***i*** must take specific compulsory courses belonging to set $T_i$.
3. Course ***j*** needs $p_j$ number of students to start the course, no more or less.
4. Courses in set ***U*** are offered in both semesters and can be taken at most twice. All other courses are offered in one of the semesters and can be taken at most once. Taking a course in set ***U*** twice will contribute two units towards the total number of courses taken by the student.

Please design an algorithm to determine if all students can graduate.

# Solution

We will solve this problem by finding a feasible circulation in a flow network.

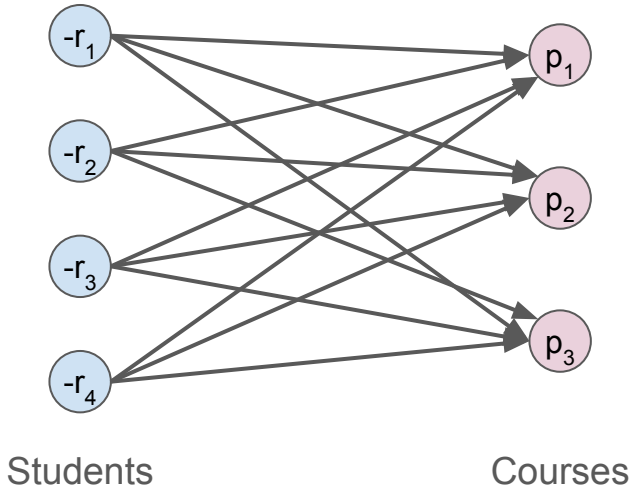
Create *n* vertices for students, and *m* vertices for courses. Add an edge from each student vertex to course vertex.



# Solution

Student *i* must take $r_i$ number of courses. Therefore, put a demand of $-r_i$ on student vertex *i*.

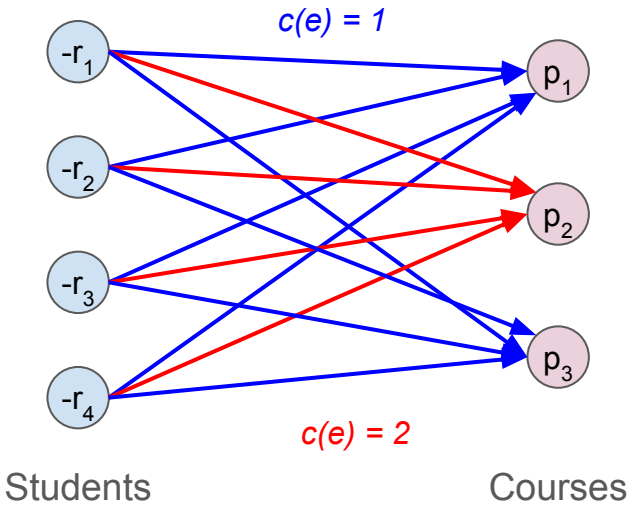
Course *j* needs $p_j$ number of students to start the course. Therefore, we put a demand of $p_j$ on course vertex *j*.



# Solution

Courses in set *U* can be taken at most twice. All other courses can be taken at most once.

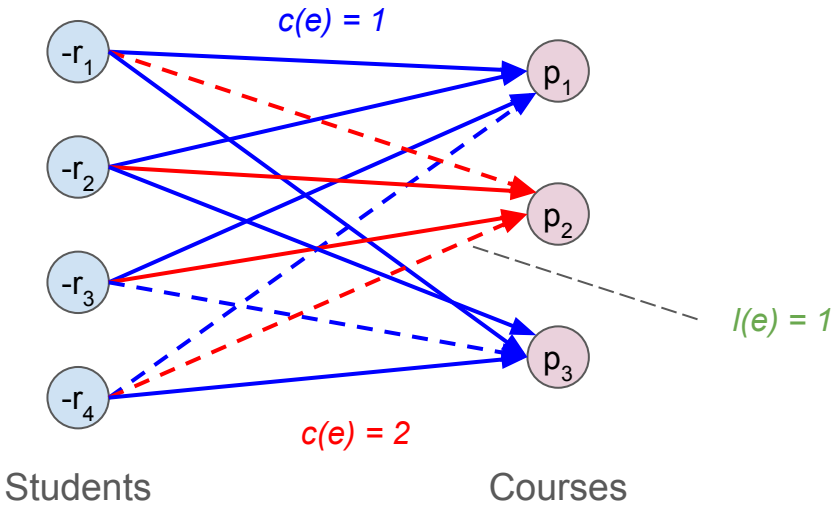
Therefore, we put a capacity of 2 on edges adjacent to a course vertex $\in U$, and a capacity of 1 on all other edges.



# Solution

Student *i* must take specific compulsory courses belonging to set $T_i$.

We put a lower bound of 1 on edges (*i*, *j*) if $j \in T_i$



# Solution

We solve the circulation with demands and lower bounds using the method we discussed earlier