1) given $G(V, E)$, $w(e)$ could be -ve, no -ve cycle count of shortest path b/w s & t = ?

since there are negative edge weights we should use bellman-foud as discussed in class.

basic step of Bellman ford is relaxation in which for each edge $u \xrightarrow{w_e} v$ if $d(v) > d(u) + w_e$ then $d(v)$ is updated with $d(u) + w_e$. After n steps all edges are relaxed those participate in shortest path b/w s to t.

we sample all those edges which are relaxed (have $d(v) = d(u) + w_e$) let's have these edges form a set R

To get the count between s to t path in G we need to find the count of paths in $G' = G(V, R)$

Since there is no neta negative edge weight cycle G' will be a DAG (no cycle) and we could simply get the count of path from s to t in topological sort. $S, V_1, V_2, V_3 \ldots V_k, t$

take an array count and initialize $count[s] = 1$

for $i = 1 \ldots k+1$

$$count[V_i] = \sum_{\substack{j \text{ st} \\ e_{ij} \in R}} C[V_j]$$

Proof of correctness:

if an edge $e(x, y)$ is relaxed then it is part of shortest path $\Rightarrow$ $s \to x$ & $y \rightsquigarrow t$ is also shortest.

let's have one path st $e_1\ e_2\ e_3 \ldots e_k$ where $e_i = (x_i, x_{i+1})$

$$\Rightarrow \quad D[x_{i+1}] - D[x_i] = w_{ei}$$

Adding this for above path we have $D_{k+1}[x_{k+1}] - D[x_1]$ as $x_1 = s$ & $x_{k+1} = t$ for shortest path this value will be.

$$D[x_{k+1}] - D[x_1] = D[t] - D[s] \to 0$$

$\Downarrow$ length of shortest path.

hence we just need to count total no of paths using edges in G'

2.

Case A. given $G(V,E)$ $\forall e$ $w_e \geq 0$ except 1 edge find the shortest s t path in $O(E \log V)$

since algorithm's complexity should be $O(m \log n)$ we are bound to use dijkstra. Also there is only one -ve edge hence we could possibly have only two cases.

case 1 : -ve edge w_e^- is in shortest path.

case 2 : -ve edge w_e^- is not in shortest path.

we just need to take both combination and find the min^m path length calculated by dijkstra

let $G' = G(V, E/e^-)$ $e^-(u,v)$ = -ve weight edge.

for case 2 we run dijkstra from s to t in G'

for case 1 we ran dijkstra i) from s to u
ii) from v to t

Shortest path $D[t] = \min \begin{cases} \text{dijkstra}(s,t) \text{ in } G', \\ \text{dijkstra}(s,u) \text{ in } G' + w(u,v) + \text{dijkstra}(v,t) \text{ in } G' \end{cases}$

CASE B: when k edges are of -ve edge weight

This case will also solved using above defined approach as k is constant. we need to take all possible combi permutation of $1, 2, \ldots k$ edges that has negative edge weight.

let w' is the set of edges with negative weight then

$G'(V, E') = G(V, E/w')$

$w' = \{e_1\ e_2\ e_3\ \ldots\ e_k\}$ wh. $e_i = (u_i, v_i)$

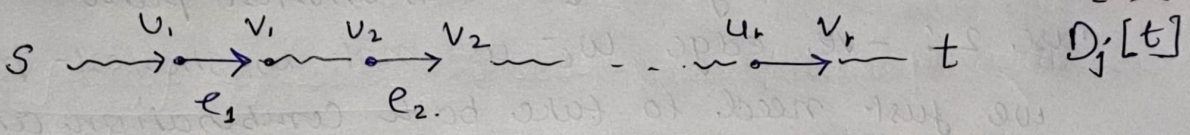
i) calculate $D[t]$ by calling dijkstra on G' from s to t

ii) calculate $D[v_i]$ by calling dijkstra on G' from v_i to t

$\rightarrow O((k+1) m \log n) \simeq O(m \log n)$

similar to case A we can have multiple -ve edges which could participate in shortest path let these are j edges hence → $e_1\ e_2\ e_3 \ldots e_j$

length of path = $D[u_1] + we_1 + D_1[u_2] + we_2 + D_2[u_3] + we_3 \ldots D_j[t]$



we just need to know the all j edges and their permutation to calculate the shortest path. which will be $2^k . k! \simeq O(1)$

we don't have any other option to check all such combination and report the ~~min~~ min weight shortest path.

3). Design efficient algorithm to find second MST.

our algorithm consist of generating second MST from MST itself.

The main idea to take MST T and remove one of its edge (w_T) and add another edge from $G-T$ which ~~makes~~ minimizes $w_T - w_{G-T}$. w_{G-T}

Algorithm :

Given : $G(V, E)$

output : second MST {set of edges}

1. Sort all edges & put in another array E'
2. generate MST as ~~for~~ kruskal's Algorithm
3. mark all edges in T True in ~~of~~ Tree array $|E|$
4. Create an array of size $|E|$ for effective weight (EW)

5. for all edges e in E' ——— O(E)
   if Tree[e] = True.
      set EW[e] = ∞
   elseif Tree[e] = False
      $Add e in T and find the higest weight edge e' in that cycle ——— O(V)
      Set EW[e] = $W_{e'} - W_e$
   end if
6. for all edges e in E'
      get min EW[e] value.
7. add e_{min} to T and remove e'_{min} from T.
8. return T

Proof of correctness:

for all non Tree edges we are checking one by one effective weight. is the non tree edge which has minimum of its value ⇒ adding that edge to T and removing the higest weight edge from cycle formed.

This algorithm also shows the second MST is only ~~def~~ one edge different from actual MST.

Complexity

Steps 1 to 4 could be performed in O(E log V).
Step 5 could be performed in O(VE)
Step 6, 7 could be performed in O(E).

total worst case complexity is O(VE).

Also one we get the zero EW[k] for any k we could stop as we are able to get edge k, by adding of which we can get second MST of same weight to MST.

4) A: array of size n
B: " " " minimize $\sum_{i=1}^{n} |A[i] - B[\sigma(i)]|$

most naive way to solve this problem is permute all combination for B and evaluate objective function Although this approach is $O(n!)$ complexity which is too much.

one greedy approach to solve this problem could be. sort both A & B (in whatever order) and then calculate $\sum_{i=1}^{n} A[i] - B[i]$.

Algorithm :

1. sort A and generate σ_A & $\mathbb{R}^n$ of size $|n|$ where σ_A : permutation of A sorted.
2. sort B and generate σ_B similarity
3. generate σ from σ_A & σ_B as:

$$\sigma = \sigma_B [\sigma_A]$$

Analysis

σ basically map the ordering of B to the disbibution of A that leads to the ith element of A_sorted to be subbacted by. ith element in B_sorted. since modulus of this difference is taken we can get the optimal solution by this greedy approach

5) you are given N, 2 operations i) increment i
ii) double i
find minimum number of such operation

In this problem there are some interesting points that can be noated.

Let's try to move from N to 1 and define the new operation o' that are inverse of above two

i) decrement i
ii) half i

Note that we are dealing with integers, hence we could only apply opera^{n} ii (half i) for even number. Also Note that - operat^{n} (ii) also gives us log based complexity

we have inversed the problem because being greedy on this modified problem will give efficient algorithm.

we are going greedy to apply operat^{n} (ii) wherever possible

for better representation we represent our number in binary. to apply both operations from o' we can write as.

MSB - LSB

if number is even: (divide by 2) if last bit [LSB] is 0 { remove LSB }

if number is odd: (decrement by 1) if LSB is 1 { change LSB to 1 }

we will first convert N to binary & then apply ② till only single bit (1) is left

if N has a number of 0 bits and b number of 1 bits then total number of operation should be $a + 2(b-1)$ Answer

* for each 1 bit we need 2 operat^{n} (except for MSB)

6) given Array A of length N.
find minimum number of disjoint subarray which cover all positive elements in it

Algorithm given:

Let A[i] be the 1st positive integer from left. Starting from A[i] find the largest index j s.t. the subarray A[i] to A[j] is positive. take this subarray and then solve the remaining problem for A[j+1] ... A[n]. using same algo.

we need to cover all positive numbers
⇒ we could remove negative numbers which are making sum most negative in subarray.

to find such numbers we see the number i such that $\sum_{j=0}^{i} j \geq 0$ & $\sum_{j=0}^{i+s} j < 0$ where i^+ : denote the $\{i+1, \ldots, n\}$

$\sum_{j=0}^{i} j$ is basically moving sum of array which helps us to locate element to be removed. (above specified equation (highlighted in pink line)).

Step 1: There should be an optimal solution which also pick the number i such that

$\sum_{j=0}^{i} j \geq 0$ and $\sum_{j=0}^{k} j < 0$ for $k = \{i+1, i+2, \ldots n\}$

Let's assume optimal solution does not pick the number at i specified above.
In that case it should chose number ~~less~~ which is at position less than i (as more than i will never be positive) leaving larger problem to solve. moreover it needs to leave the number at i too as moving sum will be going to be negative

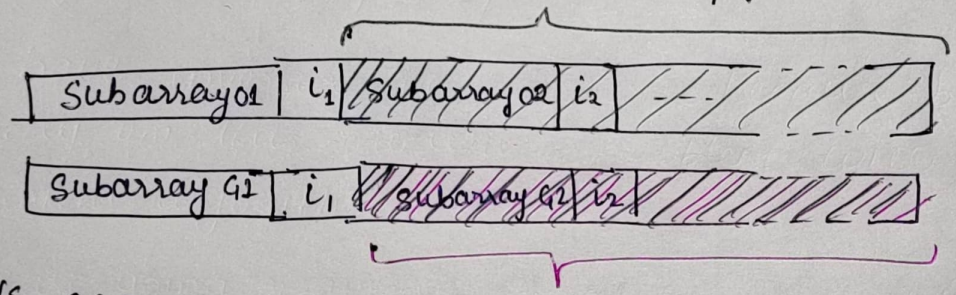
at that point again (after removing the smaller subarray

⇒ solution O (optimal) is unnecessary cutting the positive subarray which is not expected from optimal

⇒ ~~it shows~~ optimal solution will agree to greedy soln on this step.

Step 2 : proof by induction :

let's Assume optimal algorithm agrees with greedy for selection of $i_2, i_3, \ldots i_k$ wher i_T is the T^{th} number skipped/removed by algo. we need to use induction on T.



~~let's assume~~

induction hypothesis : subarray O_i = subarray G_i $\forall i \geq 2$.

from the step 1 we know that subarray O_1 is equal to subarray G_1.

⇒ Subarray O_i = Subarray G_i $\forall i = \{1, \ldots k\}$.

⇒ greedy algorithm $G \cong$ optimal algorithm O.

from step 1 and 2 we can say that our approach to get maximum possible subarray which is positive is optimal.