Amy Seo

Dis. 1D
10-11:50 AM
Song

1. Observations / Examples

n = # of final projects
g = grading scale of each project
H = total hours available to work on the projects

Ex: n=2, H=3

# of projects \ # of hrs available	0	1	2	3
0	0	0	0	0
1	0			
2	0			

- maximizing grade is enough to maximize average because n is constant throughout the algorithm
- whenever adding a new project, compare all possibilities of j hours spent on current projects and the leftover hours spent on one less total amount of projects.

Algorithm:

- initialize array OPT of size n x H
- set all OPT(i,0) = OPT(0,H) = 0
- For i = 1 to i = n:
  - For j = 0 to j = H:

$$OPT(i,j) = \max_{0 \le k \le j} f_i(k) + OPT(i-1, j-k)$$ for all hours k=0 to k=j

- return OPT(n,H)

Proof:

- Base case: when there are n=0 projects, the only score possible is 0. Thus, we have found the max score of 0 for all 0 to H hours.
- Inductive Hypothesis: Suppose there are n=k projects and H hours and the maximum average grade for $0, 1, \dots, j, \dots H$ spent on this project is x_j at hour j.
- Inductive Step: Suppose there are n=k+1 projects
  - There are H+1 possibilities: spend 0, 1, 2, ... or H hours on the k+1th project and H, H-1, H-2, ... or 0 hours on the rest of the k projects respectfully
  - Since we have stored all maximum average grades for 0 to H hours spent for k projects, we can get the maximum grade by calculating all possible hours spent on the n^{th} project, and all leftover hours being spent on n-1 projects, which has already been calculated in our inductive hypothesis.
  - This gives us a recurrence relation of

$$OPT(i,j) = \max_{0 \le k \le j} f_i(k) + \underbrace{OPT(i-1, j-k)}_{x_{j-k} \text{ by inductive hypothesis}}$$

∴ By induction, our algorithm will always find the maximum average grade possible.

Time Complexity:

- initialize $OPT(i,0) = OPT(0,j) = 0$ → $O(n+H)$ time
- outer loop = $O(n)$
- inner loop = $O(H)$
- recurrence relation = $O(H)$

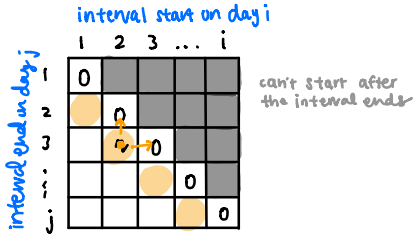
} $O(nH^2)$

Total = $O(nH^2 + n + H)$
= $O(nH^2)$

2. Observations / Examples

- single transaction: $1000(P(j)-P(i))$
  $P[i,j]$ = profit of buying on day i and selling on day j
- best single transaction to do on an interval (i,j): $Q[i,j]$
  - can buy on i, sell on j
  - can buy on i, sell on $j-1$
  - can buy on $i+1$, sell on j

} each has it's own max already calculated (recurrence relation)



Algorithm:

- initialize an $n \times n$ array $Q[i,j]$ to hold max profit for a single transaction that occurs during days i to j
- For $i=1$ to $i=n-1$
  - $Q[1,i+1] = 1000(p(i+1) - p(i))$
- For $i=2$ to $i=n-1$
  - For $j=1$ to $j=n-i$
    - let $k = i+j$
    - $Q[j,k] = \max(1000(p(k)-p(j)), Q[j+1,k], Q[j,k-1])$
- Initialize a $k \times n$ array $M[m,d]$ to hold the max profit by an m-shot strategy on days 1 to d
- set $M[m,0] = M[0,d] = -\infty$
- For $m=1$ to $m=k$:
  - For $d=0$ to $d=n$:
    - If $d<2m$
      $M[m,d] = -\infty$
    - Else
      $M[m,d] = \max_{1 \le i \le j \le d} Q[i,j] + M[m-1, i-1]$
- return $\max_{0 \le m \le k} M[m,n]$ and trace back through each entry in table to get k-shot strategy

Proof:

Part 1 - Finding max profit over an interval

- Base case: Buying and selling shares on the same day (an interval of [i,i]) gives a max profit of $0.
- Inductive Hypothesis: Suppose Q[i,j-1] is the max profit for a single transaction that occurs over the interval of days i to j-1
- Inductive Step: Suppose our interval starts on day i to day j.

  Case 1: The max single shot in this interval is buying on day i and selling on day j
  1000 (p(j) - p(i))

  Case 2: The max single shot in this interval occurs in the interval [i+1, j] where the shares are sold at <u>least</u> a day later
  Q[i+1, j]

  Case 3: The max single shot in this interval occurs in the interval [i, j-1] where the shares are sold at ~~least a day sooner~~ <u>least</u>
  Q[i, j-1]

  Out of these options, we choose the maximum, which will always give the maximum single shot on the interval [i,j]

$$Q[i,j] = \max\left(1000(p(j)-p(i)),\ Q[i+1,j],\ Q[i,j-1]\right)$$

Part 2 - Finding max profit over d days with maximum of m shots possible

- Base case: When there is a maximum of 0 shots, the maximum profit is 0 no matter the number of days.
- Inductive Hypothesis: Suppose we are given a maximum of m shots that can be performed over d days and the maximum profit is M[m,d].
- Inductive Step: Suppose we have a maximum of m+1 shots over d+1 days.
  - we want to add one more shot (one more transaction) that would maximize the previous m shots + this shot.
  - this 'max shot' occurs on days i to j where $1 \le i < j \le d+1$ and all previous shots must be completed by day i.
  - Thus, the recurrence relation is proven:

$$M[m+1, d+1] = \max_{1 \le i \le j \le d+1} Q[i,j] + M[m, i-1]$$

  as we know that M[m, i-1] must be the maximum profit for performing m shots over i-1 days by the inductive hypothesis and Q[i,j] is the maximum single shot over the interval i to j as proven in part 1.

∴ This algorithm always finds the maximum profit that can be made with k shots over n days.

Time Complexity:

- Finding max single transactions (filling out Q) = $O(n^2)$
- Finding max profit for m shots over d days (filling out M) = $O(k \cdot n^2)$
- Total = $O(kn^2)$

3. Observations

- n = # of precincts
- m = # of voters per precinct
- $A_1, \ldots, A_i, \ldots, A_n$ number of votes in precinct i for party A
- $B_1, \ldots, B_i, \ldots, B_n$ number of votes in precinct i for party B
- need to make 2 districts of size $\frac{n}{2}$
- total # of voters = $m \cdot n$
  - ↳ each district has $\frac{mn}{2}$ voters, so need $\frac{mn}{4}$ voters for majority
- subproblem:
  - ① add precinct i with A_i votes for party A to D_1
    - total # precincts $+= 1$
    - # precincts in D_1 $+= 1$
    - total # A votes in D_1 $+= A_i$
    - total # A votes in D_2 stays same
  - ② add precinct i with A_i votes for party A to D_2
    - total # precincts $+= 1$
    - # precincts in D_2 $+= 1$
    - total # A votes in D_2 $+= A_i$
    - total # A votes in D_1 stays same

* let D_1 = district 1, D_2 = district 2

- final problem:
  - n total precincts
  - $n/2$ precincts in D_1 (implies $n/2$ precincts in D_2)
  - total # A votes in $D_1 \geq \frac{mn}{4}$
  - total # A votes in $D_2 \geq \frac{mn}{4}$

} 4 parameters for DP array

Algorithm:

- initialize an empty $n \times n \times mn \times mn$ array V with all values initially set to FALSE except $V[0,0,0,0] = \text{TRUE}$
- For $i=1$ to $i=n$
  - For $j=1$ to $j=n$
    - For $k=0$ to $k=mn$
      - For $\ell=0$ to $\ell=mn$

$$V[i,j,k,\ell] = V[i-1, j-1, k-A_i, \ell] \text{ or } V[i-1, j, k, \ell - A_i]$$

$V[i,j,k,\ell]$	$V[i-1, j-1, k-A_i, \ell]$	$V[i-1, j, k, \ell-A_i]$
total i precincts	total $i-1$ precincts	total $i-1$ precincts
j allocated to D_1	$j-1$ allocated to D_1	j allocated to D_1
k votes for A in D_1	$k-A_i$ votes for A in D_1	k votes for A in D_1
ℓ votes for A in D_2	ℓ votes for A in D_2	$\ell - A_i$ votes for A in D_2

- if $V[n, \frac{n}{2}, a, b] = \text{TRUE}$ where $a \geq \frac{mn}{4}$ and $b \geq \frac{mn}{4}$ return TRUE
- else return FALSE

Proof:

- Base case: when there are 0 precincts with 0 of them allocated to D_1, it is possible for both districts to have 0 votes, so $V[0,0,0,0]$ is TRUE.
- Inductive Hypothesis: Suppose there are $n=i$ precincts with j precincts allocated to D_1, with k votes for party A in D_1 and ℓ votes for party A in D_2. $V[i,j,k,\ell]$ = TRUE as this scenario is possible.
- Inductive Step: Suppose there are $n=i+1$ precincts.

Case 1: The $i+1^{th}$ precinct is allocated to District 1

- Then it must be possible to have a total of i precincts with $j-1$ precincts allocated to District 1 with $k-A_{i+1}$ votes for party A in D_1 and ℓ votes for party A in D_2 in order for it to be possible to have a total of $i+1$ precincts with j precincts allocated to D_1 with k votes for party A in D_1 and ℓ votes for party A in D_2.

Case 2: The $i+1^{th}$ precinct is allocated to District 2

- Then, it must be possible to have a total of i precincts with j precincts allocated to D_1 with k votes for party A in D_1 and $\ell - A_{i+1}$ votes for party A in D_2 in order for j precincts out of $i+1$ precincts to be allocated for D_1 with k votes for party A in D_1 and ℓ votes for party A in D_2.

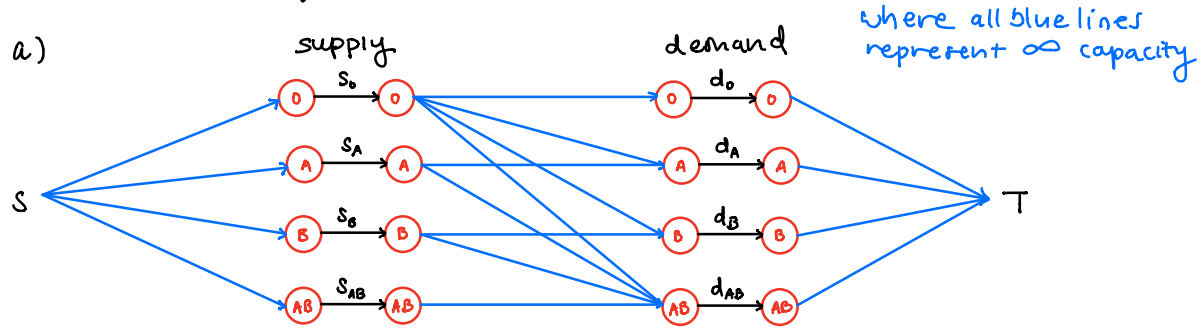
- There two cases can be summarized in this recurence relation: $V[i,j,k,\ell] = V[i-1,j-1,k-A_i,\ell]$ OR $V[i-1,j,k,\ell-A_i]$
  - $V[i-1,j-1,k-A_i,\ell]$: allocate i^{th} precinct to District 1
  - $V[i-1,j,k,\ell-A_i]$: allocate i^{th} precinct to District 2
- Then, we simply search our matrix for when there are n total precincts with half of them allocated to D_1 with a majority ($\frac{mn}{4}$) of votes for party A in both districts: $V[n, \frac{n}{2}, a, b]$ where $a \geq \frac{mn}{4}$ and $b \geq \frac{mn}{4}$
  - If such a true entry exists, the gerrymandering is possible.
- By induction, our algorithm always correctly finds whether or not it is possible to have a gerrymandering

Time Complexity:

- Setting all values in the matrix: $O(n \cdot n \cdot mn \cdot mn)$
- finding if $V[n, \frac{n}{2}, a, b]$ where $a \geq \frac{mn}{4}$ and $b \geq \frac{mn}{4}$ is true: $O(mn \cdot mn)$
- Total runtime = $O(m^2n^4)$

4. Observations / Diagram

a)



Algorithm:

- create an unlimited source S that outputs flow into the blood supply nodes that have a capacity s_i (the number of supply)
- connect the output of the O supply node to all demand nodes
- connect output of A supply node to demand nodes A and AB
- connect output of B supply node to demand nodes B and AB

} all with infinite capacities

- connect output of AB supply node to demand node AB
- each demand node has its capacity d_i
- connect output of all demand nodes to the end node T with infinite capacities
- run Ford & Fulkerson's
- if max flow = the sum of all demand return True; else return False

Proof: Suppose by contradiction, our algorithm returns false when the supply does indeed meet the demand.

- Then, there must be at least one more unit of flow (blood) that can be sent from the supply to the demand
  - note that O donates to all blood types,
    A donates to A & AB
    B donates to B & AB
    AB donates to AB
- This is a contradiction as there must be no more paths from S to T in order for Ford & Fulkerson's to terminate
  - Ford & Fulkerson's always finds the max path

∴ By contradiction, our algorithm always figures out if the blood supply meets the demand.

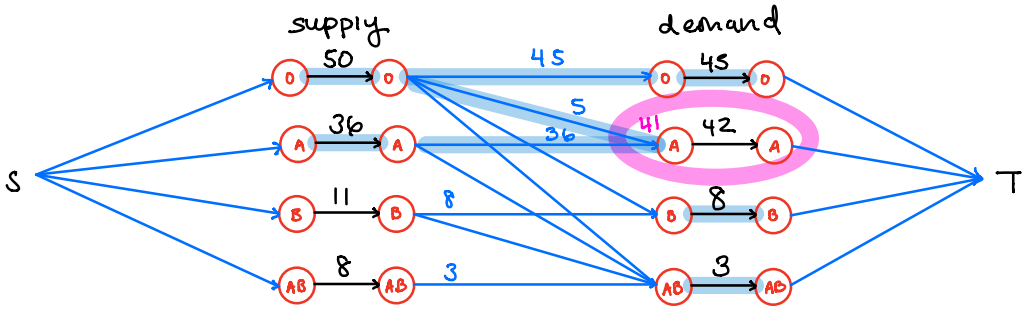
Time Complexity:

- Ford & Fulkerson's = $O(|f| \cdot (n+e))$ ← edges = 17 always
  - always 16 nodes
  - supply: 4 blood types x 2 (to hold capacity)
  - demand: 4 blood types x 2 (to hold capacity)
  - since $n+e$ are always constant (always only 4 blood types)

the overall runtime = $O(|f|)$

b)

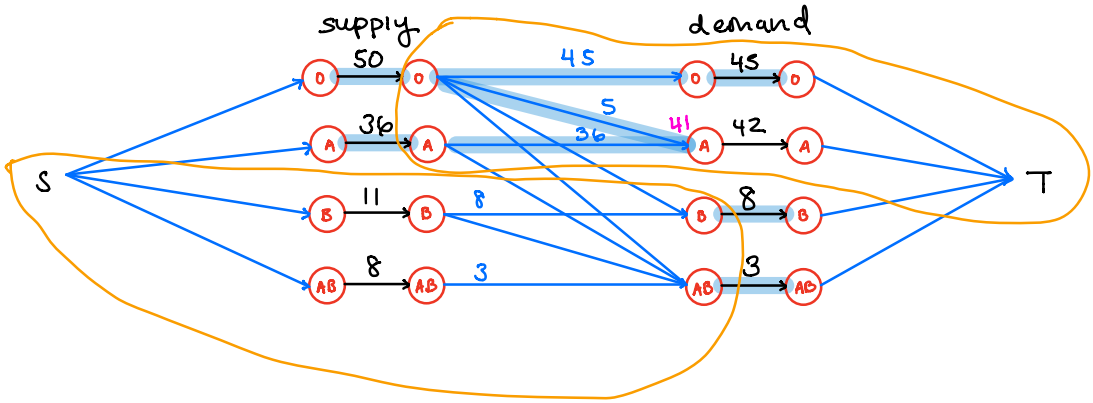
blood type	supply	demand
O	50	45
A	36	42
B	11	8
AB	8	3



max allocation:

	supply	allocation	demand met?
O:	50	45 to O 5 to A	O: 45 ✓
A:	36	36 to A	A: 42 ✗ only have 41
B:	11	8 to B	B: 8 ✓
AB:	8	3 to AB	AB: 3 ✓

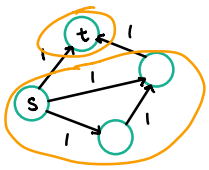
- In this example, there is not enough blood to satisfy all the demand
- All patients cannot receive blood because the total demand is 98 but our max flow from Ford & Fulkerson's is 97, as seen in this min-cut:



max flow = 50+36+8+3 = 97
but demand = 98

- Explanation w/ our algorithm terminology: All patients cannot receive blood because the total demand for blood types O and A is 45+42=87, but there is only 50+36=86 supply available.

5. Observations / Diagram



- min-cut
- since each edge has weight 1, we can reduce max flow by at most k

Algorithm:
- using Ford + Fulkerson's, find the max flow
- with the resulting residual graph, find the min-cut by finding the edges from reachable nodes from S to non-reachable nodes from S
- delete k edges in the min-cut

Proof: Suppose by contradiction, there is some set F' containing k edges that when removed from G, reduces the max flow more than our algorithm does, which produces a set F of k edges to remove.
- since all edge capacities are 1, there must be an edge e' that is part of the min-cut that can be removed to reduce the max flow, since the min-cut value is the max flow
- This is a contradiction as our algorithm would have removed these edges, and removing such an additional edge would go past the given parameter k.
  - there is no "better" edge to remove in the min-cut as all capacities are 1.

∴ By contradiction, our algorithm always removes k edges such that the max flow is minimized as much as possible.

Time Complexity:
- Ford + Fulkerson's = $O(|f| \cdot (n+e))$
- finding min-cut + removing edges: $O(n+e)$ at max
- Overall = $O(|f| \cdot (n+e))$

6. Observations / Example

index	0	1	2	3	4	5	6	7	8
nums	8	9	6	4	5	7	3	2	4
greater	1	2	2	2	4	4	4	4	6
less than	1	1	3	3	3	3	5	5	5

8 < 9 > 4 < 7 > 2 < 4 length = 6

input	2	4	5	7	5	5	7	1	2	3
greater	1	2	2	2	2	2	4	4	6	6
less than	1	1	1	1	3	3	3	5	5	5

2 < 7 > 5 < 7 > 2 < 3 length = 6

Algorithm: let nums be the input array

- initialize G[] to be an empty array w/ same length as input to hold subsequence where last element in its subsequence is > the second to last element
- initialize L[] to be an empty array w/ same length as input to hold subsequence where last element in its subsequence is < the second-to last element
- set G[0] = L[0] = 1
- For i = 1 to i = length of nums - 1:
  - if nums[i] > nums[i-1]:
    - G[i] = L[i-1] + 1
    - L[i] = L[i-1]
  - else if nums[i] < nums[i-1]:
    - L[i] = G[i-1] + 1
    - G[i] = G[i-1]
  - else
    - G[i] = G[i-1]
    - L[i] = L[i-1]
- set l = max (last element in G, last element in L)
- initialize empty array output[]
- repeat until l = 0:
  - if l is in G:
    add max of elements in input with a corresponding value of l in G to beginning of output
  - if l is in L:
    add min of elements in input with a corresponding value of l in L to beginning of output
  - set l = l - 1
- return output

Proof by Induction

- base case: when there is only one element, the max alternating subsequence is itself.
- Inductive Hypothesis: Suppose there are n numbers in the input array and the max alternating subsequence has length l
- Inductive Step: Suppose there are n+1 numbers in the input array Let us denote the $n+1^{th}$ number as X.

  Case 1: X is greater than its previous number
  - Then the length of the max alternating subsequence is the value of the most recent decreasing number + 1
    G[i] = L[i-1] + 1

  Case 2: X is less than its previous number
  - Then the length of the max alternating subsequence is the value of the most recent increasing number + 1
    L[i] = G[i-1] + 1

∴ Our algorithm always finds the length of the maximum alternating subsequence, and the rest of our algorithm traces back to the subsequence itself to print it out.

Time complexity:

- Iterating through the array to find the length of the max. alternating subsequence $= O(n)$
- Tracing back to find the subsequence itself $= O(n)$
- Total $= O(n) + O(n) = O(n)$