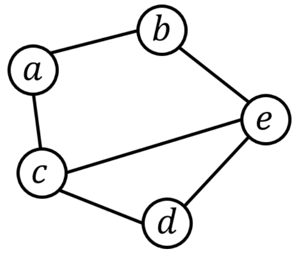
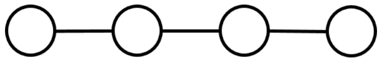
COSC 311: ALGORITHMS
HW4: ALGORITHMIC PARADIGMS AND NETWORK FLOW
Solutions

Problem 1: Independent sets.
Let $G = (V, E)$ be an undirected graph with n nodes. A subset of nodes is called an *independent set* if no two nodes in the subset are connected by an edge. For example, here's a graph:

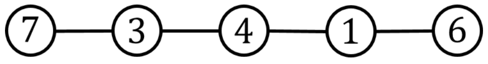


The set of nodes $\{a, e\}$ is an independent set because there is not an edge between nodes a and e, but the set of nodes $\{b, c, d\}$ is not an independent set because there is an edge connecting nodes c and d. In general, it is hard to find large independent sets in graphs efficiently (and we will discuss this problem in much more detail in a few weeks!) But in some cases—namely if the graph is "simple" enough—it can be done.

We say that a graph is a *path* if its nodes can be written as $v_1, v_2, \ldots, v_n$ and there is an edge between every pair of vertices v_i and v_{i+1}. For example, here's a path graph on 4 nodes:



We will also associate with each node a positive integer weight (note that these weights are attached to *nodes*, not to edges). For example, here's a path graph with node weights:



In the *path weighted independent set* problem, we have:

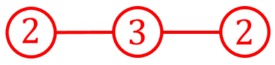
Input: an undirected, unweighted path graph $G = (V, E)$ with $|V| = n$ nodes, and integer weight $w_i > 0$ on each node i

Output: a subset of nodes S such that no two nodes in S have an edge between them, and the total weight of the nodes in S, $W = \sum_{i \in S} w_i$, is maximized

(a) Give an example of a path graph on which the following "heaviest first" greedy algorithm does not produce the optimal solution.

```
HeaviestFirst(G = (V,E))
    S = {}
    while V isn't empty
        pick the node v in V with maximum weight
        add v to S
        delete v and its neighbors from V
    return S
```

Solution: Here's a graph:

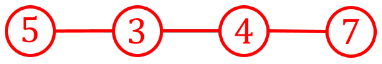


The "heaviest first" algorithm says to choose the middle node, which has value 3. A better solution is to choose the first and third nodes; this solution has total value 4.

(b) Give an example of a path graph on which the following "even-odd" algorithm does not produce the optimal solution.

```
EvenOdd(G = (V,E))
    let S1 be the set of all nodes in an even position in the path
    let S2 be the set of all nodes in an odd position in the path
    return whichever of S1 and S2 has greater total weight
```

Solution: Here's a graph:



The "even-odd" algorithm says to choose the second and fourth nodes, which have combined weight 10. A better solution is to choose the first and last nodes, which have combined weight 12.

(c) Write a recurrence that expresses the optimal solution for a path graph of n nodes in terms of smaller subproblems.

Solution: We'll define subproblem i to be the problem of choosing the highest-valued set of nodes from among the first i nodes (labeled from left to right). We can choose either to include node i in our solution, or not. If we include node i then we're not allowed to include node $i-1$, so we need to know the best value we can get when considering just the first $i-2$ nodes. If we don't include node i then the best we can do is whatever our best solution was for the first $i-1$ nodes. We end up with the recurrence:

$$V(i) = \max\{w_i + V(i-2), V(i-1)\}$$

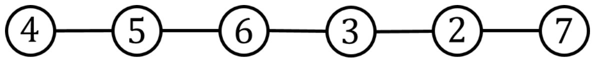
(d) Translate your recurrence from part (c) into a dynamic programming algorithm that solves the path weighted independent set problem optimally; that is, your algorithm should take as input a weighted path graph, and output an independent set of nodes with maximum total weight. Your algorithm's runtime should be polynomial in n and should not depend on the node weights.

Solution: We'll fill in an array of subproblem solutions. One way to do this is bottom-up, solving smaller subproblems before larger subproblems:

```
IndependentSet(G = (V,E))
    V[0...n]: a new array to store best values
    I[0...n]: a new array to store whether to include node i
    set V[0] = 0
    for i = 1 to n
        if w_i + V[i-2] > V[i-1]
            set V[i] = w_i + V[i-2]
            set I[i] = include
        else
            set V[i] = V[i-1]
            set I[i] = skip
    // now we need to construct the set of nodes to include
    set S = {}
    set i = n
    while i > 0
        if I[i] = include
            set S = S + {i}
            set i = i - 2
        else set i = i - 1
    return S
```

Other variations are possible. For example, this could be written as a top-down recursive algorithm with memoization, and the reconstruction step also could be written recursively in a separate method.

(e) Consider the following path graph:



Explain how your algorithm goes about finding the optimal solution for this graph. What solution does your algorithm produce?

Solution: We begin with the first node; including it gives us value 4 and skipping it gives value 0, so we record `V[1] = 4` and `I[1] = include`.

Next we consider the second node. If we include it, we must skip the first node, giving us total value 5; if we skip it we use whatever the best solution was for node 1. Including the second node is better, so we record `V[2] = 5` and `I[2] = include`.

Third node: If we include it, we must skip the second node, giving us value 6 + `V[1]` = 10. If we skip it, we take the solution from `V[2]`, which is 5. 10 is better, so we record `V[3] = 10` and `I[3] = include`.

Fourth node: If we include it, we get value 3 + `V[2]` = 8. If we skip it, we take the solution from `V[3]`, which is 10. 10 is better, so we record `V[4] = 10` and `I[4] = skip`.

Fifth node: If we include it, we get value 2 + `V[3]` = 12. If we skip it, we take the solution from `V[4]`, which is 10. 12 is better, so we record `V[5] = 12` and `I[5] = skip`.

Sixth node: If we include it, we get value 7 + `V[4]` = 17. If we skip it, we take the solution from `V[5]`, which is 12. 17 is better, so we record `V[6] = 17` and `I[6] = include`.

We now retrace our steps to reconstruct the optimal set of nodes to choose. Starting at the end, we see that `I[6] = include`, so we add node 6 to our solution and proceed backwards to node 4 (since we cannot include node 5, having chosen node 6). We see that `I[4] = skip`, so we do not add node 4 to our solution and proceed backwards to node 3. We see that `I[3] = include`, so we add node 3 to our solution and proceed backwards to node 1. We see that `I[1] = include`, so we add node 1 to our solution and proceed backwards to node -1, at which point we are done. Our solution ends up being nodes 1, 3, and 6, which has value 4 + 6 + 7 = 17.

(f) What is the runtime of your algorithm, in terms of n?

Solution: Our algorithm involves a for loop in lines 5-11 that runs for n iterations. Inside this loop we have constant time work: comparisons, conditionals, variable assignments, array accesses. So the entire loop runs in time $\Theta(n)$. The reconstruction also involves a loop (lines 15-19) that starts at $i = n$ and runs down to $i = 0$, which takes time $\Theta(n)$ in the worst case; again the work inside the loop takes constant time. So the overall runtime of our algorithm is $\Theta(n)$.

Problem 2: Maximizing profit.

Suppose we are given a list of prices for a given stock for n consecutive days, where the days are numbered $i = 1, 2, \ldots, n$. For each day, we are given the price per share p_i for the stock on that day (and we'll assume that the price stayed the same all day). We want to determine on what day i to buy the stock and on what day $j > i$ to sell the stock in order to maximize our profit per share, p_j/p_i. For the *stock profit* problem, we have:

Input: a list of prices $p_i > 0$ for each day $i = 1, 2, \ldots, n$

Output: a pair of days (i, j) on which to buy and sell the stock, such that $j > i$ and the profit p_j/p_i is maximized (if there is no possible way to make a profit, i.e., if $p_j/p_i \leq 1$ for all days i, j, then we should output "no solution" instead)

(a) There are several possible ways to solve this problem. We've discussed three different algorithmic paradigms: divide-and-conquer, greedy, and dynamic programming. If you were to design an algorithm to solve this problem, which paradigm would you choose? Explain why you believe your chosen paradigm is the best fit for this problem. You may want to discuss whether this problem satisfies the properties needed to apply a paradigm and attain an optimal solution, what runtimes you might be able to achieve, or anything else that might factor into your decision about what paradigms are well-suited to this problem.

Solution: A naive solution to the problem involves considering all pairs of days (i, j), computing the profit for each pair, and choosing the best one. There are $\Theta(n^2)$ possible pairs, and it will take constant time (just a division) to compute the profit for a single pair, so it'll take us time $\Theta(n^2)$ to run this naive solution. Our goal in coming up with a different algorithm must therefore be to beat a $\Theta(n^2)$ runtime.

The greedy paradigm probably isn't the right choice for this problem. It's hard to imagine what a greedy choice would look like. We can see in the example in part (c) that greedily choosing the day with the lowest price as the buy date won't work, nor will greedily choosing the day with the highest price as the sell date.

One way to think about what paradigms are reasonable is to try to identify problems we've seen before that feel similar to this problem, and think about how we solved those problems. This problem has a somewhat parallel structure to the largest sum subsequence problem discussed in class. In that problem, we had an array of numbers and were trying to find the contiguous sub-array with the largest sum. At a high level, both the LSS problem and the stock profit problem involve choosing a starting point and a later ending point within the array. We used a divide-and-conquer approach to solve the LSS problem, so perhaps a similar approach will work here. Like with that problem, we can imagine splitting our days down the middle and then considering the possibilities for where our solution (i.e., the best days on which to buy and sell) could fall. And like with the LSS problem, we have three options: (1) it's best to both buy and sell in the left half, (2) it's best to both buy and sell in the right half, or (3) it's best to buy in the left half and sell on the right half.

Dynamic programming is another reasonable choice for this problem. Even though a greedy algorithm won't work here, the optimal substructure does hold. Here's why: suppose the optimal solution for all days 1 through n is to buy on day i and sell on day j. This must also be the best pair of buy and sell days for the range of days from 1 through j. If not, there would be some better pair i' and j', with $1 \leq i' < j' \leq j$, that gave a larger profit. But since i' and j' fall between days 1 and j, they also fall between days 1 and n, so the pair (i', j') would have given a better solution to the original problem. However we don't know how to choose j, so we'll need to consider all possibilities. This will lead us to a dynamic programming solution; the hope is that by defining the problem in a way that involves overlapping subproblems, we'll be able to get away with doing less work than the $\Theta(n^2)$ naive solution.

(b) Give an algorithm that fits within the paradigm you chose in part (a) to solve the stock profit problem optimally for any input.

Solution:

Divide-and-conquer version

We have three options for where the optimal buy and sell dates could lie:

1. Buy and sell in the first half of the days
2. Buy and sell in the second half of the days
3. Buy in the first half and sell in the second half

For the first two options, we are left with a smaller version of the same type of problem, so we can solve these cases recursively. For the third option, note that if we're going to buy in the first half and sell in the second half, the best way to do this will be to buy on the day in the first half with the lowest price, and sell on the day in the second half with the highest price. Our algorithm is (to be called initially on `StockProfitDandC(p,1,n)`):

```
StockProfitDandC(p, lo, hi)
    // we'll use 0 to indicate "no solution"
    if hi <= lo, return (0,0)
    set mid = (lo + hi)/2
    set (li,lj) = StockProfitDandC(p, lo, mid)
    set (ri,rj) = StockProfitDandC(p, mid+1, hi)
    set (ci,cj) = CrossingProfit(p, lo, mid, hi)
    // return whichever option yields the highest profit
    if p[li]/p[lj] > p[ri]/p[rj]
        if p[li]/p[lj] > p[ci]/p[cj], return (li,lj)
        else return (ci,cj)
    else if p[ri]/p[rj] > p[ci]/p[cj], return (ri,rj)
    else return (ci,cj)

CrossingProfit(p, lo, mid, hi)
    set small = lo
    for i = lo+1 to mid
        if p[i] < p[small], set small=i
    set large = mid+1
    for i = mid+2 to hi
        if p[i] > p[large], set large=i
    return (small,large)
```

Dynamic programming version

We are trying to find the highest profit across all days 1 through n. Starting at the end, at day n, we have two options: either sell on day n, or sell on some earlier day. If we sell on some earlier day, the best profit we can achieve by day n is the same as the best profit we can achieve by day $n-1$. If we sell on day n, the best profit we can achieve involves buying on the day before day

n lowest price. This suggests the following recurrence for $V(j)$, the maximum profit achievable when selling on day j or earlier:

$$V(j) = \max\{V(j-1), \frac{p_j}{\min_{i<j} p_i}\}.$$

We translate this into an algorithm as follows:

```
1  StockProfitDP(p)
2      V[0...n]: a new array, stores the best profit when selling
                 at or before day i
3      M[0...n]: a new array, stores the index of the min price
                 at or before day i
4      S[0...n]: a new array, stores the pair of days on which to
                 buy and sell to get the profit stored in V
5      set V[0] = 0, set M[0] = infty, set S[0] = (0,0)
6      for j = 1 to n
7          M[j] = M[j-1]
8          if p[j] < p[M[j-1]]
9              set M[j] = j
10         V[j] = V[j-1]
11         S[j] = S[j-1]
12         if p[j]/M[j-1] > V[j]
13             set V[j] = p[j]/p[M[j-1]]
14             set S[j] = (M[j-1], j)
15     return S[j]
```

Unlike with Bellman-Ford, it takes a constant extra amount of space to store the best (buy,sell) pair for each i, so we do this as we're going to make it easy for us to find the solution without having to retrace our steps at the end.

(c) Here's a list of prices:

i	1	2	3	4	5	6	7	8
p_i	4	24	2	16	20	1	3	9

Explain how your algorithm goes about finding the optimal solution for this particular input. What solution does your algorithm find?

Solution:

Divide-and-conquer version

We begin by consider the range of days from 1 to 8. In lines 5 and 6 we recursively solve the problems on the left half (days 1 through 4) and on the right half (days 5 through 8). The best solution on the left half is to buy on day 3 and sell on day 4 (profit $16/2 = 8$), and the best solution on the right half is to buy on day 6 and sell on day 8 (profit $9/1 = 9$) The `CrossingProfit` method then finds that the min on the left side is price 2 on day 3, and the max on the right side is price 20 on day 5 (profit $20/2 = 10$). 10 is the best of these three options, so we return (3,5) (buy on day 3, sell on day 5) as the optimal solution.

How did we find the best solutions for days 1 through 4 and for days 5 through 8? For days 1 through 4, we recursively solve the problems on the left half (days 1 through 2) and on the right half (days 3 through 4). The best solution on the left half is to buy on day 1 and sell on day 2 (profit $24/4 = 6$), and the best solution on the right half is to buy on day 3 and sell on day 4 (profit $16/2 = 8$). The `CrossingProfit` method then finds that the min on the left side is price 4 on day 1, and the max on the right side is price 16 on day 4 (profit $16/4 = 4$). 8 is the best of these three options, so we return (3,4) (buy on day 3, sell on day 4) as the optimal solution for days 1 through 4.
We could write out the steps similarly for the other branches of the recursion.

Dynamic programming version
We start by setting `V[0] = 0`, `M[0] = infty`, and `S[0] = (0,0)`. We then fill in position 1 for each array: we set `M[1] = 1` (since p[1] $= 4 < \infty$), then we set `V[1] = 0` and `S[1] = (0,0)` (since `p[j]/M[j-1]` $= 4/\infty = 0$ is not better than the 0 profit we found from the nonexistent day 0). On the next iteration, we fill in `M[2] = 1` (our previous minimum, `p[M[1]] = 4`, is less than `p[2] = 24`). This time since `p[2]/p[M[1]]` $= 24/4 = 6 < 0 =$ `V[1]`, we set `V[2] = 6` and `S[2] = (1,2)`. Proceeding in this manner, we end up with the table:

i	1	2	3	4	5	6	7	8
p_i	4	24	2	16	20	1	3	9
`M[i]`	1	1	3	3	3	6	6	6
`V[i]`	0	6	6	8	10	10	10	10
`S[i]`	(0,0)	(1,2)	(1,2)	(3,4)	(3,5)	(3,5)	(3,5)	(3,5)

(d) What is the runtime of your algorithm, in terms of n?

Solution:

Divide-and-conquer version
We'll write a recurrence for the runtime. We split our problem of size n into two subproblems, each of size $n/2$. Our "divide" step takes constant time (just computing the midpoint). Our "combine" step is the call to `CrossingProfit`, plus the sequence of comparisons in lines 9-13. `CrossingProfit` involves two for loops, the first running from `lo+1` to `mid` and the second from `mid+2` to `hi`; this gives a total of `hi-lo` iterations, which at the top level is n iterations. There's a constant amount of work inside each iteration. So `CrossingProfit` takes linear time. Our recurrence is thus:

$$T(n) = 2T(n/2) + cn$$

At this point, we might recognize that this is the same recurrence we've seen for, e.g., mergesort. Or we can apply the Master Theorem. We have $a = 2$ and $b = 2$, so $n^{\log_b a} = n^{\log_2 2} = n$, and $f(n) = cn$, so $f(n) \in \Theta(n^{\log_b a})$. This puts us in Case 2 of the Master Theorem, so we conclude that $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n \lg n)$.

Dynamic programming version
Our for loop in lines 6-14 runs for n iterations. On each iteration, we do a constant amount of work: indexing into arrays several times, comparisons, variable assignments, divisions. The overall runtime is $\Theta(n)$.

Problem 3: Blood transfusions.
Suppose you work for a hospital, and your job is to determine whether the current supply of blood will be enough to meet the projected demand over the next week for blood transfusions.

Unfortunately, it's not as simple as checking whether the number of liters currently available is greater than the number of liters required: people have *blood types* that constrain the types of blood they can receive. A person's blood supply has certain antigens present, and a person cannot receive blood that contains a particular antigen if their own blood does not have this antigen. Concretely, blood is divided into four types: A, B, AB, and O. Blood of type A has antigen A, blood of type B has antigen B, blood of type AB has both antigens, and blood of type O has neither. So a patient with type-A blood can receive only blood types A or O in a transfusion, a patient with type-B blood can receive only blood types B or O, a patient with type-AB blood can receive any of the four types, and a patient with type-O blood can only receive blood type O. In the *blood transfusion* problem, we have:

Input: s_A, s_B, s_{AB}, and s_O: the supply of each of the different blood types (assume these are all positive integers), and d_A, d_B, d_{AB}, and d_O denote the demand of each of the different blood types in the next week (again, assume positive integers)

Output: "yes" or "no" to the question: is it possible to meet the demand for each blood type? And if so, how?

(a) Give an algorithm to determine if the current supply of blood is enough to meet the demand. Your algorithm should model the blood transfusion problem as a network flow problem (i.e., represent the problem using a graph, and then run a max-flow algorithm on the resulting graph). In your algorithm description, it should be clear (i) what the vertices and edges represent in the graph you create, (ii) how, after running a max-flow algorithm on the graph you created, you can determine if the current blood supply is sufficient, and (iii) if the blood supply is sufficient, how the available blood should be allocated to patients with each blood type.

Solution: (i) We will form a graph G' as follows:

- Create two nodes x_i and y_i for each blood type $i = A, B, AB, O$, where node x_i is a "supply" node and node y_i is a "demand" node.
- Create a source node s and a target node t.
- Create a directed edge from s to each node x_i with capacity $c(s, x_i) = s_i$, and create a directed edge from each node y_i to t with capacity $c(y_i, t) = d_i$.

- Create a directed edge from x_i to y_j with capacity ∞ if patients with blood type j can receive donations of type i.

(ii) Then run Ford-Fulkerson on the resulting graph to find $v(f)$, the value of the max flow. If $v(f) = \sum_i d_i$, then the available supply is sufficient to meet the demand.

(iii) The flow found by Ford-Fulkerson gives us a way of allocating blood to patients that meets the demand: if the flow along edge (x_i, y_j) is $f(x_i, y_j)$, then use $f(x_i, y_j)$ units of type-i blood for transfusions to patients with type-j blood. Our capacity constraints on edges (s, x_i) guarantee that we don't use more type-i blood than we have in the supply, and our capacity constraints on edges (y_j, t) guarantee that we don't allocate more blood to type-j patients than there is demand. Hence this is a valid allocation of blood to patients.

As a related point that isn't actually part of what you were asked to do in this question, we can also show that for any allocation of blood to patients that meets the demand, we can construct a valid flow . Let's suppose we have a valid allocation that uses $u_{i,j}$ units of type-i blood in transfusions to patients with type-j blood. Then set the flow as follows:

$$\begin{aligned} f(s, x_i) &= \sum_j u_{i,j} \\ f(x_i, y_j) &= u_{i,j} \\ f(y_j, t) &= \sum_i u_{i,j} \end{aligned}$$

The first line says that the total flow from s to x_i is equal to the total number of units of type-i blood that are used. The second line says that the flow from the type-i supply node to the type-j demand node is equal to the number of units of type-i blood that are used in transfusions to patients with type-j blood. The third line says that the total flow from y_j to t is equal to the total number of units of blood used in transfusions to patients with type-j blood.

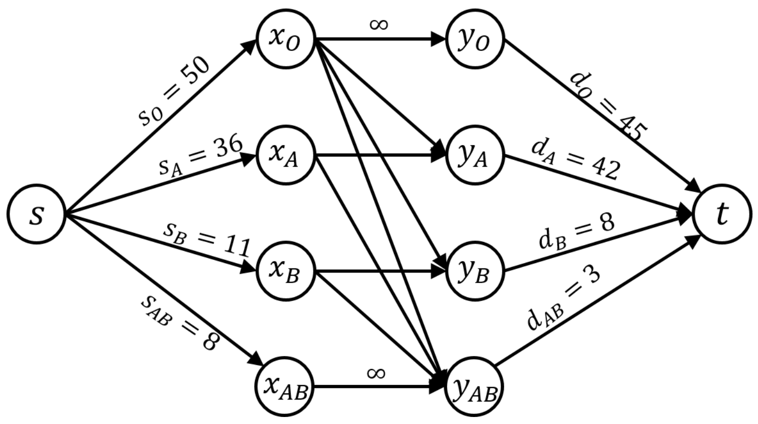
This flow satisfies the capacity constraints because in a valid allocation (1) we cannot use more units of type-i blood than we have, so $f(s, x_i) = \sum_j u_{i,j} \leq s_i$, and (2) we cannot allocate more units of blood to type-j patients than there is demand, so $f(y_j, t) = \sum_i u_{i,j} \leq d_j$. The flow also satisfies the conservation constraints because (1) at any node x_i, we have $f^{\text{in}}(x_i) = f(s, x_i) = \sum_j u_{i,j} = \sum_j f(x_i, y_j) = f^{\text{out}}(x_i)$, and (2) at any node y_j, we have $f^{\text{in}}(y_j) = \sum_i f(x_i, y_j) = \sum_i u_{i,j} = f(y_j, t) = f^{\text{out}}(y_j)$.

(b) Consider the following example:

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

Show how your algorithm works on this example. You should draw a picture of the graph you create and describe one possible run of Ford-Fulkerson on the graph.

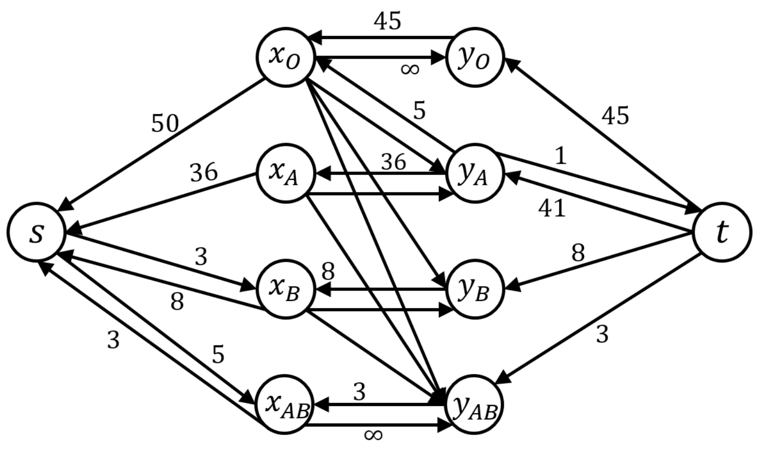
Solution: Here is the graph we create:



Here is one possible run of Ford-Fulkerson:

1. Send 3 flow along path s, x_{AB}, y_{AB}, t.
2. Send 8 flow along path s, x_B, y_B, t.
3. Send 36 flow along path s, x_A, y_A, t.
4. Send 45 flow along path s, x_O, y_O, t.
5. Send 5 flow along path s, x_O, y_A, t.

At this point, here is the residual graph:



We can see that there are no paths from s to t in the residual graph, so our run of Ford-Fulkerson is complete. The max flow that we found has value 97.

(c) Your algorithm from part (a) should output two things: a yes/no answer and (if yes) an allocation of blood supply to patients that meets the demand for all blood types. Suppose you run your algorithm and learn that the available blood supply is not enough to meet the demand in the next week. Even though your algorithm outputs the answer "no," you could imagine outputting the latter part of the solution anyway. In this case, this piece of the solution doesn't give an allocation that meets the full demand. But it isn't entirely meaningless. What do you learn from the output if it was not possible to meet the full demand?

Solution: If it is not possible to meet the full demand, the output of our algorithm tells us how to meet as much of the demand as we can: use $f(x_i, y_j)$ units of type-i blood for transfusions to patients with type-j blood.

In the example from part (b), it is not possible to meet the full demand. The total demand is 98, and the value of the max flow that we found is 97. Intuitively, the reason we can't meet the full demand is that if we look at the subset of blood types O and A, we see that the total demand for these blood types is $45 + 42 = 87$. However, the total supply that can be used in transfusions to either blood type is only $50 + 36 = 86$, so we will not be able to meet the demand.

In this case, the resulting flow tells us to use the following allocation:

- Use 45 units of type-O blood in transfusions to type-O patients.
- Use 5 units of type-O blood in transfusions to type-A patients.
- Use 36 units of type-A blood in transfusions to type-A patients.
- Use 8 units of type-B blood in transfusions to type-B patients.
- Use 3 units of type-AB blood in transfusions to type-AB patients.

While this allocation does not meet the full demand (we are short 1 unit of blood for type-A patients), it does give us a way to meet as much of the demand as possible.