Exercises (strongly recommended, do not turn in)

1. (DPV textbook, Problem 7.11.) Find the optimal solution to the following linear program.

$$\begin{aligned} \max\ & x + y \\ 2x + y &\leq 3 \\ x + 3y &\leq 5 \\ x, y &\geq 0 \end{aligned}$$

Solution: the extreme points in the feasible region are $(0, 0)$, $(0, 5/3)$, $(4/5, 7/5)$, and $(3/2, 0)$, of which $(4/5, 7/5)$ has the highest objective function value at $11/5$.

2. (DPV textbook, Problem 7.21.) An edge of a flow network is called *critical* if decreasing the capacity of this edge results in a decrease in the maximum flow. Give an efficient algorithm that finds a critical edge in a network.

Solution:

(a) **Algorithm Description in English**
First, run the max flow algorithm. If an edge is critical, it must be filled to capacity, otherwise it would be possible to reduce its capacity to the amount of flow going through it without affecting the maximum flow. Then, for each edge (u, v) filled to capacity in the residual graph, run DFS and see if there is a path from u to v. If there isn't, then that edge is a critical edge.

(b) **Pseudocode**

```
define getCriticalEdge(G)
residualGraph = FordFulkerson(G)
for edge (u, v) in residualGraph such that
(v, u) is not in residualGraph and (u, v) is not in G:
  DFS(residualGraph, u)
  if u has no path to v:
    return (u, v)
  return null
```

(c) **Proof of Correctness**
If an edge is not full when max flow is achieved, it is possible to reduce its capacity to the amount of flow going through it without affecting the maximum flow. By running the Ford-Fulkerson algorithm, we can determine which edges are full when the maximum flow is achieved. It is not enough for the edge to be full, however. We must check that the flow cannot be re-routed along another path in the residual graph. Suppose (u, v) is an edge that is full after the Ford- Fulkerson algorithm is run. If there is a path from u to v not using (u, v), then if the amount of flow allowed through (u, v) is decreased by 1, then the flow can just be routed from u to v through the other path, without affecting the max flow. If there is

no path from u to v, then there is no way to re-route the flow that would have gone through (u, v), so the maximum flow must decrease. Therefore, full edges are only critical edges if there does not exist any path from u to v. Running DFS finds if there exists a path between u and v.

(d) **Running Time Analysis**
To run this algorithm, first you must run the Ford-Fulkerson algorithm, which takes $O(VE^2)$. Next, you must run DFS on at most $O(V)$ vertices, taking $O(V(V + E))$ time. This term is dominated by the running time of the Ford-Fulkerson algorithm, so the overall running time is $O(VE^2)$.

3. (DPV textbook, Problem 7.23.) The *vertex cover* of an undirected graph $G = (V, E)$ is a subset of the vertices which touches every edge- that is, a subset $S \in V$ such that for each edge $u, v \in E$, one or both of of u, v are in S.
Show that the problem of finding the minimum vertex cover in a bipartite graph reduces to maximum flow. (Hint : Can you relate this to the minimum cut in a related network?)

Solution: Call the two partitions of the bipartite graph S_1 and S_2 . Add vertices s and t. Draw an edge from s to every vertex in S_1 and from every vertex in S_2 to t. By definition of bipartite graph, we know all edges in E will connect some vertex u in S_1 with some vertex v in S_2 . For every edge in E, draw an edge into the flow graph with weight 1 from the vertex in S_1 to the vertex in S_2 . Set the capacity of all edges in the graph to be 1.

The maximum matching of a graph is the largest set of edges that can be selected so that no two edges in the set touch the same vertex. This problem can be solved by using max flow, by setting up the same graph described above and running the Ford-Fulkerson algorithm. The maximum matching will be the set of all edges between S_1 and S_2.

The minimum vertex cover can be solved the same way. Solve for the maximum matching. Consider all of the vertices in the graph besides s and t. Call the set of these vertices that are an endpoint of an edge in the maximum matching A, and call the set of vertices that are not endpoints of any of the maximum matching edges B.

There can be no edges between distinct elements in B. Assume for contradiction that there was. That edge could then have been added to increase the size of the maximum matching, because it runs between two vertices that were untouched by the maximum matching edges. This contradicts our assumption that the matching found was the maximum matching, so it is impossible to have edges between distinct elements of B.

Therefore, all edges in the graph must either be included in the maximum matching or must be be- tween a vertex that is an endpoint of a maximum matching edge and a vertex in B. Note that it is also impossible for a maximum matching edge to have both endpoints have edges to elements in B, otherwise the size of the matching could have been increased by removing the edge from the maximum matching and adding the two edges from its endpoints to the vertices in B. Therefore, at most one endpoint vertex of every maximum matching edge can have an edge to a vertex in B.

Therefore, when constructing the minimum vertex cover, it is sufficient to take every maximum match- ing edge and choose one of its endpoints for the minimum vertex cover. Specifically, if one of the endpoints has and edge to a vertex in B, choose that vertex. Otherwise, it doesnt matter which one you choose.

Because an endpoint was selected from every edge in the maximum matching, then the vertex cover must cover all of the maximum matching edges. Furthermore, if a vertex had an edge to an element in B, then that vertex was chosen. There can be no edges between elements in B, as we proved earlier. Therefore, the vertex cover chosen in this way must cover every edge in the graph. Furthermore, it must be minimal, as removing any of the vertices would mean that neither endpoint vertex of one of the maximum matching edges was in the set of vertices, because only one endpoint vertex was chosen per maximum matching edge. This means that the resultant set of vertices would not cover every edge in the graph. Therefore, if we can solve maximum matching, we can solve minimum vertex cover. Since maximum matching is solved by max flow, then minimum vertex cover reduces to max flow.

Problems (must be written up and turned in)

1. DPV 6.2: You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \ldots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance a_n), which is your destination.

   You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the penalty for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penalty- that is, the sum, over all travel days, of the daily penalties. Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

Solution: Suproblems definition
Let $OPT(i)$ be the minimum total penalty to get to hotel i.

Recursive formulation
To get $OPT(i)$, we consider all possible hotels j we can stay at the night before reaching hotel i. For each of these possibilities, the minimum penalty to reach i is the sum of:

- the minimum penalty $OPT(j)$ to reach j,
- and the cost $(200 - (a_j - a_i))^2$ of a one-day trip from j to i.

Because we are interested in the minimum penalty to reach i, we take the minimum of these values over all the j:

$$OPT(i) = \min_{0 \leq j < i} \{OPT(j) + (200 - (a_j - a_i))^2\}$$

And the base case is $OPT(0) = 0$.

Pseudo-code

```
// base case
OPT[0] = 0
// main loop
for i = 1...n:
  OPT[i] = min([OPT[j] +(200 − (a_j − a_i))^2 for j=0...i-1])
// final result
return OPT[n]
```

Complexity
- We have n subproblems,
- each subproblems i takes time O(i)

The overall complexity is

$$\sum_{i=1}^{n} O(i) = O(\sum_{i=1}^{n}) i = O\frac{n(n-1)}{2} = O(n^2)$$

2. DPV 6.13: Consider the following game. A "dealer" produces a sequence $s_1 \dots s_n$ of "cards", face up, where each card s_i has a value v_i. Then two players take turns picking a card from the sequence, but can only pick the first or the last card of the (remaining) sequence. The goal is to collect cards of largest total value. (For example, you can think of the cards as bills of different denominations.) Assume n is even.

   (a) Show a sequence of cards such that it is not optimal for the first player to start by picking up the available card of larger value. That is, the natural greedy strategy is suboptimal.

   Solution: Consider the sequence (2, 10, 1, 1). Then the best available card at first is 2. Then the second player can pick the card 10 and will win. If the first player pick the rightmost 1 first, then the first will be able to pick the 10 and will win.

   (b) Give an $O(n^2)$ algorithm to compute an optimal strategy for the first player. Given the initial sequence, your algorithm should precompute in $O(n^2)$ time some information, and then the first player should be able to make each move optimally in $O(1)$ time by looking up the precomputed information.

   Solution: Suproblems definition
   Let $OPT(i, j)$ be the difference between:
   - the largest total the first player can obtain,
   - and the corresponding score of the second player
   when playing on sequence $s_i, ..., s_j$.

   Recursive formulation
   At any stage of the game, there are 2 possible moves for the first player:
   - either choose the first card, in which case he will gain v_i and score $-OPT(i+1, j)$ in the rest of the game,
   - or the last card, gaining v_j and $-OPT(i, j-1)$ from the remaining cards.
   Because we are interested in the largest total the first player can gain over the second, we take the maximum of these 2 values (for $i < j$):

$$OPT(i, j) = max\ \{v_i - OPT(i+1, j),\ v_j - OPT(i, j-1)\}$$

   And the base cases are: $\forall i \in \{1, ..., n\},\ OPT(i, i) = v_i$.

   Pseudo-code
   Precomputation

```
// base cases
for i = 1...n:
OPT[i, i] = v[i]
// main loop
for j = 1...n:
  for i = j...1:
    OPT[i, j] = max (v[i] - OPT(i+1,j), v[j] - OPT(i,j-1))
// final result
return OPT[1, n]
```

   Look up
   Given sequence $s_i, \dots, s_j$

$$OPT\_move[i, j] = \begin{cases} s_i, & \text{if } OPT[i, j] = v[i] - OPT(i+1, j) \\ s_j, & \text{otherwise} \end{cases}$$

Complexity
Precomputation
- We have $O(n^2)$ subproblems,
- each update takes time $O(1)$.
Therefore, the overall complexity is $O(n^2)$.

3. Every morning you purchase the daily donut from the exotic bakery Dynamic Pastries. You have a schedule over a period of n days for which the donut on day i costs p_i. They also offer a special where, on any given day, you can prepay for the next 13 days for a fixed price C. You don't want to have any prepaid days left over at the end of the n days, so you can't purchase the special if there are less than 13 days remaining. Given a set of n prices $p_1, \ldots, p_n$, the special cost C, and your insane donut addiction, you want to figure out the minimum amount of money that you need to cover the n days.

   (a) You first try a greedy approach. For any day i, you compare special price C versus the cost of purchasing individually for the next 13 days ($p_i + p_{i+1} + \cdots + p_{i+12}$). If the special is cheaper, you buy it; otherwise you buy the donut individually for that day. Give an example where this strategy leads you to spending more than needed.

   Solution: For the sequence $1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2$ and $C = 3$. The greedy solution gives a cost of 5 but waiting a day to buy the special would yield a cost of 4.

   (b) Give an efficient dynamic programming algorithm to find the optimal solution. Be sure to provide a recursive formulation and analyze the time complexity.

   Solution: One possible subproblem:

   $$\mathcal{P}(i) = \text{minimum total price of donuts ending on day } i.$$

   An important thing to note is that this subproblem assumes that day i is the last day and thus there can be no prepays left over at that point. Given that, we can then consider on that last day, is the best solution to buy the donut that day, or have already bought a donut 13 days earlier.

   $$\mathcal{P}(i) = \begin{cases} \infty & \text{if } i < 0 \\ 0 & \text{if } i = 0 \\ \min\{\mathcal{P}(i-1) + p_1, \mathcal{P}(i-13) + C\} & \text{otherwise} \end{cases}$$

   Pseudo-code:

   $\underline{\mathcal{P}(i)}$

   ```
   if i < 0:  return ∞
   if P[i] = ∞ then:
     P[i] = min{P(i − 1) + p1 , P(i − 13) + C}
   return P[i]
   ```

   $\underline{\texttt{min-price}(p_1, \ldots, p_n, C)}$

   ```
   for i = 1, ..., n do:
     P[i] = ∞
   P[0] = 0
   return P(n)
   ```

   There are n subproblems which require $O(1)$ work to solve at each step. The total running time then is $O(n)$.

4. DPV 7.5: The Canine Products company offers two dog foods, Frisky Pup and Husky Hound, that are made from a blend of cereal and meat. A package of Frisky Pup requires 1 pound of cereal and 1.5 pounds of meat, and sells for \$7. A package of Husky Hound uses 2 pounds of cereal and 1 pound of meat, and sells for \$6. Raw cereal costs \$1 per pound and raw meat costs \$2 per pound. It also costs \$1.40 to package the Frisky Pup and \$0.60 to package the Husky Hound. A total of 240,000 pounds of cereal and 180,000 pounds of meat are available each month. The only production bottleneck is that the factory can only package 110,000 bags of Frisky Pup per month. Needless to say, management would like to maximize profit.

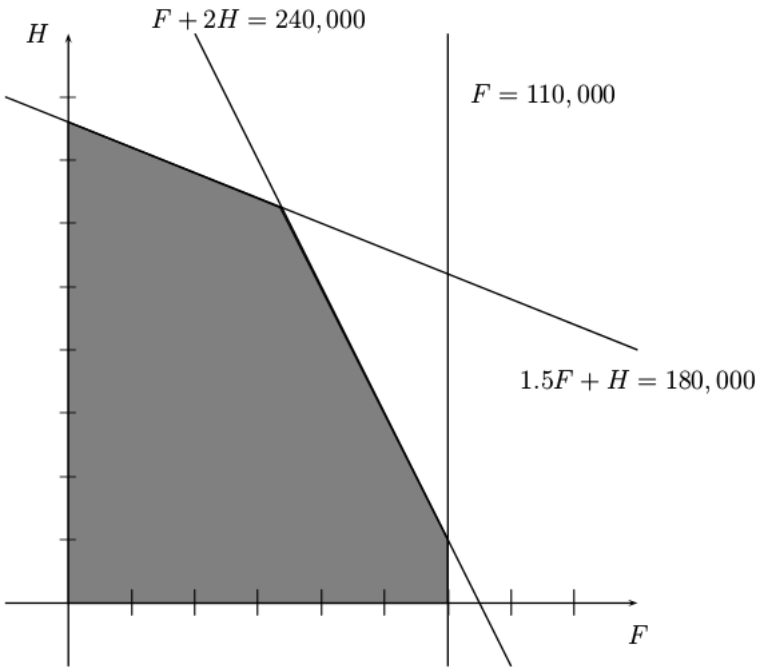
   (a) Formulate the problem as a linear program in two variables.

Solution: Maximize $1.6F + 1.4H$ subject to:

$$\begin{aligned} F + 2H &\leq 240,000 \text{ (Cereal)} \\ 1.5F + H &\leq 180,000 \text{ (Meat)} \\ F &\leq 110,000 \text{ (Production)} \\ H, F &\geq 0 \end{aligned}$$

   (b) Graph the feasible region, give the coordinates of every vertex, and circle the vertex maximizing profit. What is the maximum profit possible?

Solution:



The maximum profit possible is \$222,000 with 60000 units of F and 90000 units of H.

5. DPV 7.18: There are many common variations of the maximum flow problem. Here are four of them.

   (a) There are many sources and many sinks, and we wish to maximize the total flow from all sources to all sinks.

   Solution:
   - Let S be the set of source nodes.
   - Let T be the set of sink nodes.
   - Let I be the set of interior nodes, that are neither sources nor sinks.
   - Let $G = (V, E)$ be the original graph.

   Reduction from multi-source, multi-sink flow to single-source, single-sink flow
   Form graph $G' = (V', E')$, where $V' = \{s, t\} + V$, and

   $$E' = \{(s, a, \infty) | a \in S\} + E + \{(b, t, \infty) | b \in T\}$$

   Solve for maximum s-t flow on G'.

   (b) Each vertex also has a capacity on the maximum flow that can enter it.

   Solution:
   Let C_v denote the capacity of vertex v. (Just as C_e or $C_{(u,v)}$ shall denote the capacity of an edge.)
   Reduction from node-capacitated max-flow to simple max-flow.
   From graph $G' = (V', E)$, where

   $$V = \{a_{in} | a \in V\} + \{a_{out} | a \in V\}$$

   and,

   $$E = \{(a_{in}, a_{out}, C_a) | a \in V\} + \{(a_{out}, b_{in}, C_{(a,b)}) | (a, b) \in E\}$$

   Solve for maximum s_{in} -t_{out} flow on G'.

   (c) Each edge has not only a capacity, but also a lower bound on the flow it must carry.

   Solution:
   - Let $i : N \mapsto e$ be a bijective mapping between the natural numbers and edges in the graph.
   - Let C_i be the capacity of edge i.
   - Let L_i be the lower bound of i.
   - Let f_i be the flow along edge i.

   Maximize:

   $$\sum_{i \in (s,v)} f_i$$

   Subject to:

   $$\begin{aligned}
   f_i &\leq C_i && \forall i \in E \text{ (Capacities)} \\
   f_i &\geq L_i && \forall i \in E \text{ (Lower bounds)} \\
   \sum_{i \ into \ v} f_i &= \sum_{j \ out \ of \ v} f_j && \forall v \in V - \{s, t\} \text{ (Conservation)}
   \end{aligned}$$

   (d) The outgoing flow from each node u is not the same as the incoming flow, but is smaller by a factor of $(1 - \epsilon_u)$, where ϵ_u is a loss coefficient associated with node u.

Solution:
- Let $i : N \mapsto e$ be a bijective mapping between the natural numbers and edges in the graph.
- Let C_i be the capacity of edge i.
- Let e_v be the loss coefficient along each node, so that $outflow(v) = (1 - e_v)inflow(v)$ for all internal nodes.
- Let f_i be the flow along edge i.

Maximize:

$$\sum_{i \in (s,v)} f_i$$

Subject to:

$$\begin{aligned} f_i &\leq C_i \qquad \forall i \in E \text{ (Capacities)} \\ \sum_{i \ into \ v} (1 - e_v) f_i &= \sum_{j \ out \ of \ v} f_j \quad \forall v \in V - \{s, t\} \text{ (Lossy Conservation)} \end{aligned}$$

Each of these can be solved efficiently. Show this by reducing (a) and (b) to the original max-flow problem, and reducing (c) and (d) to linear programming.