Lelen Abeywardena	999890097	g3abeywb	lelen.abeywardena@mail.utoronto.ca
Ryan D'Souza	999999934	g3dsouza	ryry.dsouza@mail.utoronto.ca
Spencer Elliott	999992539	g3spence	spencer.elliott@mail.utoronto.ca
Ramaneek Gill	1000005754	g3gillra	ramaneek.gill@mail.utoronto.ca

1. (a) **Intuition**: adding e_1 to the minimum spanning tree (i.e. $T \cup \{e_1\}$) will introduce a cycle C. By removing the maximum-weight edge from C, we are left with a minimum spanning tree T_1.

```
1:  function MstWithEdge(G, w, T, e_1, w_1)
2:      (u, v) ← e_1
3:      P ← breadth-first search in T from u to v          ▷ P is the set of edges from u to v
4:      e_P ← maximum-weight edge in P
5:      if w_1 < w(e_P) then          ▷ If the new edge costs less than the largest edge in the existing path
6:          return T ∪ {e_1} − {e_P}          ▷ Remove the maximum-weight edge from P to break the cycle, and
7:                                                                          ▷ add e_1 to the new MST
8:      end if
9:      return T          ▷ Otherwise, ignore the new edge since it does not decrease overall cost
10: end function
```

Worst-case running time:
Let n be the number of vertices, and let m be the number of edges in G. MstWithEdge has a worst-case running time of $\mathcal{O}(n)$, as shown below:

The breadth-first search on line 3 runs in $\mathcal{O}(n)$ time, since the search will explore at most every vertex and edge in T, and there are $n-1$ edges in T (since it's a spanning tree).

Finding the maximum-weight edge in line 4 runs in $\mathcal{O}(n)$ time: iterating at most once for each edge in T.

The remaining operations run in $\mathcal{O}(1)$ time.

Proof of correctness:

P on line 3 must be the only path in T from u to v (since if there were multiple paths, a cycle would exist). Therefore adding e_1 to T would result in a cycle with the edges in P. Any edge in the cycle can then be removed to restore the minimum spanning tree.

There are two cases on line 5:

Case 1: $w_1 < w(e_P)$: the weight of e_1 is less than the maximum edge weight in P.

Then we can add e_1 to T, and remove e_P to break the cycle and restore the minimmum spanning tree property.

The new spanning tree $T_1 = T \cup \{e_1\} - \{e_P\}$ is a minimum spanning tree due to the fact that the sum of all edge weights is minimized, i.e. $w(T_1) - w(e_P)$ is minimized since $w(e_P)$ is the maximum edge weight in the cycle.

Case 2: $w_1 \geq w(e_P)$: the weight of e_1 is greater than or equal to the maximum edge weight in P.

Then no change can be made to T to reduce the sum of all edge weights, so we return the same minimum spanning tree T.

(b) **Intuition**: If $e_0 \notin T$, then $T_0 = T$ is already a minimum spanning tree. Otherwise, we must remove e_0 from T (which disconnects two components of the MST) and find the next minimum edge that can reconnect the two components. If none exists, then we must return NIL.

```
1:  function MstWithoutEdge(G, w, T, e_0)
2:      if e_0 ∉ T then          ▷ e_0 is not part of the MST for (V, E), so T remains an MST for (V, E − {e_0})
3:          return T
4:      end if
5:      (u, v) ← e_0
6:      T_0 ← T − {e_0}
7:      T_u ← edges of T_0 connected to u
8:      T_v ← edges of T_0 connected to v
9:      e_m ← NIL          ▷ e_m is the light edge crossing the cut between the two components
10:     w(e_m) ← ∞
11:     for all (x, y) ∈ E − {e_0} do
12:         if [(x ∈ T_u ∧ y ∈ T_v) or (y ∈ T_u ∧ x ∈ T_v)] and w(x, y) < w(e_m) then
```

```
13:            e_m ← (x, y)        ▷ The edge (x, y) connects the two components, and it's lighter than e_m
14:        end if
15:    end for
16:    if e_m ≠ NIL then
17:        T_0 ← T_0 ∪ {e_m}       ▷ A minimum edge e_m was found crossing the cut
18:        return T_0
19:    end if
20:    return NIL                  ▷ No edge was found that could connect the two components
21: end function
```

Worst-case running time:
Let n be the number of vertices, and let m be the number of edges in G. MSTWITHOUTEDGE has a worst-case running time of $\mathcal{O}()$, as shown below:

On lines 7 and 8, collecting the edges into sets T_u and T_v runs in $\mathcal{O}(n)$ time, since all edges and vertices in T_0 must be examined: edges + vertices $= (n-1) + n < 2n$ (since there are fewer edges than verticies in the MST).
The **for** loop on line 11 runs in $\mathcal{O}(m)$ time since it iterates at most m times (once for each edge in E).
The remaining operations (checking set membership, comparing numbers, unioning sets, destructuring edges) run in constant time.
Thus, the total run time is $\mathcal{O}(n+m)$, which is less than that of building the MST from scratch:

$$\begin{aligned} \mathcal{O}(n+m) &< \mathcal{O}(m) && \text{since } n \leq m+1 \\ &< \mathcal{O}(m \log n) && \text{running time of Kruskal's/Prim's} \end{aligned}$$

Proof of correctness:

Fact 1 (from Piazza): Removing any edge e from a tree T creates exactly two connected components.

light edge (from CLRS): An edge crossing a cut with the minimum weight of any edge crossing the cut.

The first step returns T if $e_0 \notin T$, since e_0 was not part of the MST, thus T remains a valid minimum spanning tree for $(V, E - \{e_0\})$.
Lines 8 and 9 depend on T_0 being disconnected into two components, with u and v each in separate components. Line 6 ensures this is true by removing e_0 from T. (By **Fact 1**).
In the **for** loop on line 11, we iterate over edges in $E - \{e_0\}$, which must contain an edge that crosses the cut if such an edge exists:

Case 1: $[(x \in T_u \wedge y \in T_v)$ or $(y \in T_u \wedge x \in T_v)]$ and $w(x,y) < w(e_m)$
Then one vertex of e_m must lie in the component connected to u, and the other vertex in the component connected to v. Thus (x, y) crosses the cut between the two components. Also, the weight of (x, y) is less then the previously-found edge, thus the current (x, y) is the lightest edge crossing the cut so far.

Case 2: $[\neg(x \in T_u \wedge y \in T_v)$ and $\neg(y \in T_u \wedge x \in T_v)]$ or $w(x,y) \geq w(e_m)$
Then either: (1) the vertices of (x, y) do not cross the cut, or (2) the weight of edge (x, y) is greater than the previously-found edge.
Either (1) (x, y) is not a valid edge to connect the two components, or (2) (x, y) can connect the components but has a greater weight than the already-found edge.

When the **for** loop terminates, the following is true:

$$e_m = \text{NIL} \vee e_m = (x, y) \text{ where } (x, y) \text{ is a light edge crossing the cut between } T_u \text{ and } T_v$$

If $e_m = \text{NIL}$ then line 13 was never run, so a light edge crossing the cut was not found in $E - \{e_0\}$. Therefore e_0 is the only edge in T that can connect the two components, so no MST for $(V, E - \{e_0\}$ exists.
If $e_m = (x, y)$, then e_m is a light edge in $E - \{e_0\}$ which connects the two components, thus $T_0 \cup \{e_m\}$ is a minimum spanning tree for $(V, E - \{e_0\})$.

2. **Recursive structure:**

Idea: For any block in the first row, positions $1,1$ through $1,n$, we can find the maximum possible amount of gold collected by starting at that block with a drill hardness h ($\textsc{MaxGold}(i,j,h)$) by adding $G[i,j]$ to the maximum amount of the below three blocks and reducing hardness by $H[i,j]$: $G[i,j] + \max\{\textsc{MaxGold}(i+1,j-1,h-H[i,j]), \textsc{MaxGold}(i+1,j,h+H[i,j]), \textsc{MaxGold}(i+1,j+1,h+H[i,j])\})$. Pseudocode:

function GOLDDIGGER(H, G, d):
  return $\max\{\textsc{MaxGold}(H,G,1,1,d-H[1,1]), ..., \textsc{MaxGold}(H,G,1,n,d-H[1,n])\}$
end function
function MAXGOLD(H, G, i, j, h):
  if $H[i,j] > h$ **then**
    return 0
  else if $H[i,j] = h$ **then**
    return $G[i,j]$
  else if $H[i,j] < h$ and $i < m$ **then**
    return $G[i,j] + \max\left\{\begin{array}{l}\textsc{MaxGold}(H,G,i+1,j-1,h-H[i,j]),\\ \textsc{MaxGold}(H,G,i+1,j,h-H[i,j]),\\ \textsc{MaxGold}(H,G,i+1,j+1,h-H[i,j])\end{array}\right\}$
  else ▷ $H[i,j] \leq h$ and $i = m$
    return $G[i,j]$
  end if
end function

Array subproblem values and recurrence:

We will try to memoize values produced by MAXGOLD by storing them in an array A. We'll try the same recursive top-down approach (where array values are filled from the bottom up):

Let $A[i,j,h]$ be the maximum possible amount of gold collected by starting at the block at location i,j with drill hardness h. Let $S[i,j,h]$ be the successor to block i,j along the path corresponding to $A[i,j,h]$. Then:

Base values ($i = m$):

$$A[m,j,H[m,j]] = G[m,j]$$
$$S[m,j,H[m,j]] = \text{NIL}$$

Along the bottom row of the matrix (i.e. $i = m$), each block's maximum amount of gold collected is equal to the amount of gold in that block, since we cannot drill deeper. It takes exactly $H[i,j]$ hardness to drill through just the one block.

A special value of NIL is used to terminate the path since this is the last block in any drilling path.

Recursive values ($1 \leq i < m$):

$$\begin{array}{lll} A[i,j,h+H[i,j]] = G[i,j] + A[i+1,j,h] & \text{and } S[i,j,h+H[i,j]] = j & \\ A[i,j,h+H[i,j]] = G[i,j] + A[i+1,j-1,h] & \text{and } S[i,j,h+H[i,j]] = j-1 & \text{if } j > 1 \\ A[i,j,h+H[i,j]] = G[i,j] + A[i+1,j+1,h] & \text{and } S[i,j,h+H[i,j]] = j+1 & \text{if } j < n \end{array}$$

For each layer of blocks above the bottom row, we calculate the maximum amount of gold that can be collected by adding the current block's gold to the maximum amount of the block below, requiring $h + H[i,j]$ hardness from the current block (i.e. the current block's hardness plus the hardness required for maximum gold from the block below). The same calculation is performed for blocks to the bottom-left and bottom-right if they exist.

This array definition is slightly different from the recursive structure outlined previously, since we are calculating generally the maximum amount of gold that can possibly collected for every possible hardness requirement for each block. Once the values are calculated, we can find the maximum gold amount for d by searching for the maximum value of $A[1,j,h] \forall j$ where $h \leq d$.

Algorithm:

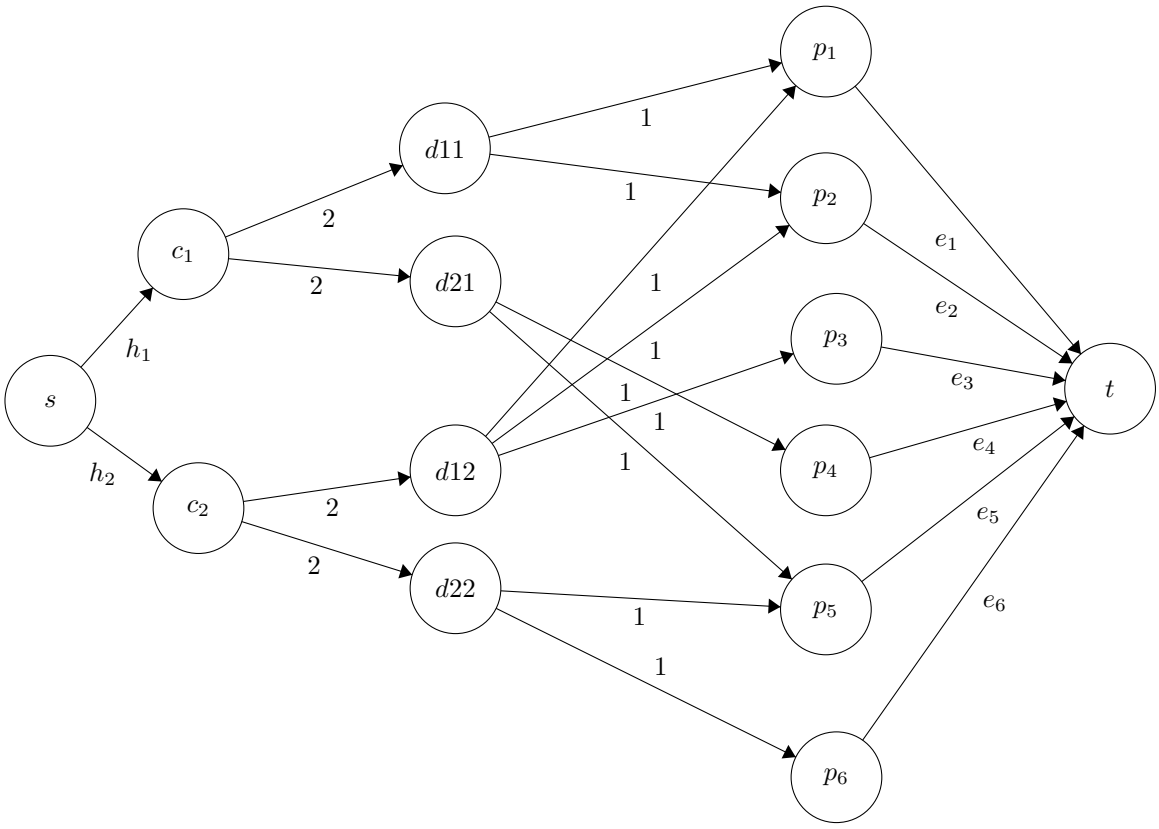
```
function DrillingPath(H, G, d):
    for all j ← 1, ..., n do                                  ▷ Populate base values for all i = m values
        A[m, j, H[m, j]] ← G[m, j]
        S[m, j, H[m, j]] ← NIL
    end for
    for all i ← m − 1, ..., 1 do
        for all j ← 1, ..., n do          ▷ For each block i, j, calculate maximum gold values based on blocks below
            for all h ∈ A[i + 1, j] do          ▷ For each hardness value previously saved for the block directly below
                A[i, j, h + H[i, j]] ← G[i, j] + A[i + 1, j, h]
                                                ▷ The maximum amount of gold that can be collected is the current block's
                                                ▷ gold, plus the maximum amount for the block below, and requires
                                                ▷ h + H[i, j] to drill (the below block's hardness requirement plus the current block's)
                S[i, j, h + H[i, j]] ← j
            end for
                                                ▷ If blocks exist to the bottom-left and bottom-right, calculate those values too
            if j > 1 then
                for all h ∈ A[i + 1, j − 1] do
                    A[i, j, h + H[i, j]] ← G[i, j] + A[i + 1, j − 1, h]
                    S[i, j, h + H[i, j]] ← j − 1
                end for
            end if
            if j < n then
                for all h ∈ A[i + 1, j + 1] do
                    A[i, j, h + H[i, j]] ← G[i, j] + A[i + 1, j + 1, h]
                    S[i, j, h + H[i, j]] ← j + 1
                end for
            end if
        end for
    end for
                ▷ Array is constructed; now we can find the maximum gold amount for d and reconstruct the path
    j′ ← NIL, h′ ← NIL
        ▷ j′ and h′ represent the optimal starting block position and valid hardness at the beginning of the path
    m ← 0                                          ▷ m represents the maximum gold amount found so far
    for all j ← 1, ..., n do                       ▷ Iterate over each of the blocks in the top row
        for all h ∈ A[1, j] do
            if A[1, j, h] > m and h ≤ d then
                                                ▷ A gold amount is found greater than m with a valid hardness requirement
                j′ ← j
                h′ ← h
                m ← A[1, j, h]
            end if
        end for
    end for
    if j′ = 0 then                                 ▷ No block position was found, so no optimal path exists
        return ∅
    end if
    P ← {j′}    ▷ Build the path (sequence of block positions) by finding each successor until the terminating NIL
    while (s ← S[1, j′, h′]) ≠ NIL do
        P ← P ∪ {s}
    end while
    return P
end function
```

3. Let h_j denote the max number of examinations CPO j can invigilate.
Let c_j denote CPO j
Let e_i denote the number of examinations held at exam period i +10%.
Let p_i denote the pool for exam period i.
Let $d_l j$ denote day l for CPO j

```
function AssignCPOs(DT, A, M):
    Output = []
    V = {s, c_1, ..., c_n, d_1 1, ..., d_k n, p_1, ..., p_m, t}
    E = {(s, c_1) : h_1, ..., (s, c_n) : h_n, (p_1, t) : e_1, ..., (p_m, t) : e_m}   ▷ Create a network from the data
    for all j ← 1, ..., n do
        for all l ← 1, ..., k do                                                      ▷ go through every day for cpo j
            c(c_j, d_l j) = 2
            for all i ←each period in d_l j do                                        ▷ for the exam periods in that day
                if p_i in A[c_j] then                                                 ▷ if the cpo is available for that period
                    c(d_l j, p_i) = 1
                end if
            end for
        end for
    end for
    N' = FORD-FULKERSON(G = (V, E))
    for all i ← 1, ..., m do
        if f(p'_i, t') == e'_i then
            for all j ← 1, ..., n do
                let l be the day of period/pool p'_i
                if f(d'_l j, p'_i) == 1 then
                    Output ← (c_j, p_i)                                               ▷ Output is pairs of CPO, pool
                end if
            end for
        else
            return false                                                              ▷ no solution
        end if
    end for
    return Output
end function
```

The main idea behind this problem is to convert it into a network flow and run this network through ford-fulkerson to get a max-flow network, from which we can extract the CPO, period pairs.
For each edge capacity from s to c_j is the max number of exams each c_j can invigilate, this will ensure no CPO invigilates more exams than they are allowed to.
Each CPO gets his own set of dates, this will make it possible for us to enforce the 2/day constraint so the edge from c_j to $d_l j$ has capacity 2.
Then since each CPO has his own day, when assigning capacities from the day to the pool we can check if the CPO is allowed to invigilate that exam period, if not we do not put and edge there/ capacity = 0. if he is allowed then we put an edge with capacity 1 from the day to the pool.
The edge from the pool to t will have capacity = # of inviligators needed for period + 10% which is what e_i represents.

Example:



Runtime $O(n * k * 3 + m * n + VE^2)$ where n is the number of CPOs and k is the number of days and m is the number of examination periods, VE^2 is the runtime of the Ford-Fulkerson algorithm.

Correctness

For any flow f in N', $c(c_j, d_l j) = 2$ this guarantees a CPO does not invigilate more than 2 exams a day.
Since c_j denotes CPOs $d_l j$ will denote CPOs after the two/day constraint is applied.
For any flow f in N', $P_i = \{l, j : f(d_l j, p_i) = 1\}$ are outputs with no CPO j in more than h_j pools.
Total size of all pools = $|f|$
So max total size $\geq$ max flow.
For any pool $p_i, \ldots, p_n, f(d_l j, p_i) = 1$ iff $c_j \in p_i$. $f(s, c_j)$ = number of p_i containing c_j, $f(p_i, t)$ = size of the pool.
If $|f|$ = total size of all pools so max flow $\geq$ max total size of all pools.