1. (a) Let D_0 = "On input $x \in \mathbb{N}$, output TRUE iff x is even."
   Then, $\overline{D_0}$ = "On input $x \in \mathbb{N}$, output TRUE iff x is odd."
   Also, $D_0 \leqslant_p \overline{D_0}$ with the reduction function $f(x) = x + 1$:
   - clearly, $f(x)$ is computable in polytime;
   - also, x is even iff $f(x) = x + 1$ is odd, i.e., $x \in D_0$ iff $f(x) \in \overline{D_0}$.

   (b) Yes, $D_0 \in NP$ because $D_0 \in P \subseteq NP$.

   (c) **Conclusion:** $NP = coNP$ (this implies that D_1 is $coNP$-complete).
   Justification: For every $D \in NP$, $D \leqslant_p D_1$ (because D_1 is NP-complete) so $D \leqslant_p \overline{D_1}$ (by transitivity of $\leqslant_p$), which implies that $D \in coNP$ (because $\overline{D_1} \in coNP$). Hence, $NP \subseteq coNP$. For every $D \in coNP$, $\overline{D} \in NP$ (by definition of $coNP$) so $\overline{D} \leqslant_p D_1$ (because D_1 is NP-complete). But then $\overline{D} \leqslant_p \overline{D_1}$ (by transitivity of $\leqslant_p$), which is equivalent to $D \leqslant_p D_1$, and this implies that $D \in NP$ (because $D_1 \in NP$). Hence, $coNP \subseteq NP$.

2. **GOLDDIGGER $\in$ *NP*:** The following algorithm verifies GOLDDIGGER in polytime.

   VERIFYGD(h, g, H, G, c):
       # c is a sequence of integers $j_1, j_2, \ldots, j_\ell$
       return TRUE iff:
           $\ell \leqslant m$
           $1 \leqslant j_k \leqslant n$ for $k = 1, 2, \ldots, \ell$
           $j_{k-1} - 1 \leqslant j_k \leqslant j_{k-1} + 1$ for $k = 2, 3, \ldots, \ell$
           $H[1, j_1] + H[2, j_2] + \cdots + H[\ell, j_\ell] \leqslant h$
           $G[1, j_1] + G[2, j_2] + \cdots + G[\ell, j_\ell] \geqslant g$

   Clearly, VERIFYGD(h, g, H, G, c) = TRUE for some c iff (h, g, H, G) is a yes-instance for GOLDDIGGER. Also, VERIFYGD runs in polytime: each arithmetic operation requires at most polytime and there are a linear number of arithmetic operations performed.

   GOLDDIGGER is *NP*-hard: SUBSETSUM $\leqslant_p$ GOLDDIGGER through the following reduction function.

   On input $\big(S = \{x_1, x_2, \ldots, x_m\}, t\big)$, output $h = g = t$ and
   - $H[1,1] = G[1,1] = 0$; $H[1,2] = G[1,2] = x_1$;
   - $H[2,1] = G[2,1] = 0$; $H[2,2] = G[2,2] = x_2$;
   - $\ldots$;
   - $H[m,1] = G[m,1] = 0$; $H[m,2] = G[m,2] = x_m$.

   Clearly, (h, g, H, G) can be computed in polytime from (S, t).

   Also, suppose S contains some subset S' whose sum is exactly t, Then consider the drilling path defined as follows, for $k = 1, 2, \ldots, m$:

$$j_k = \begin{cases} 1 & \text{if } x_k \notin S', \\ 2 & \text{if } x_k \in S'. \end{cases}$$

   $j_1, \ldots, j_m$ is a valid drilling path ($1 \leqslant j_k \leqslant 2$ for $k = 1, 2, \ldots, m$ and $j_{k-1} - 1 \leqslant j_k \leqslant j_{k-1} + 1$ for $k = 2, 3, \ldots, m$). Moreover, $H[k, j_k] = G[k, j_k] = 0$ when $x_k \notin S'$ and $H[k, j_k] = G[k, j_k] = x_k$ when $x_k \in S'$, so $H[1, j_1] + \cdots + H[m, j_m] = \sum_{x \in S'} x = t \leqslant h$ and $G[1, j_1] + \cdots + G[m, j_m] = \sum_{x \in S'} x = t \geqslant g$.

   Finally, suppose $j_1, \ldots, j_\ell$ is a drilling path such that $H[1, j_1] + \cdots + H[\ell, j_\ell] \leqslant h = t$ and $G[1, j_1] + \cdots + G[\ell, j_\ell] \geqslant g = t$. Then $H[1, j_1] + \cdots + H[\ell, j_\ell] = G[1, j_1] + \cdots + G[\ell, j_\ell] = t$ (because $H[i, j] = G[i, j]$ for all i, j). This means $S' = \{G[k, j_k] : G[k, j_k] > 0\}$ is a subset of S whose sum is exactly t.

3. Suppose that GDD(H,G,h,g) is an algorithm that solves the GoldDigger decision problem in polytime. We write an algorithm to solve the GoldDiggerOpt optimization problem.

GDO(H,G,h):
  # Compute an upper bound B on the maximum amount of gold possible:
  # simply add up the maximum gold amount on each level.
  $B \leftarrow \sum_{j=1}^{m} \max(G[j,1], G[j,2], \ldots, G[j,n])$
  Binary search in the range $[0,B]$ to find the maximum g with GDD(H,G,h,g) = True.
  # Now, "eliminate" individual blocks from consideration, one by one.
  # Loop Invariant: GDD(H,G,h,g) = True.
  for $i \leftarrow 1,2,\ldots,m$:
    for $j \leftarrow 1,2,\ldots,n$:
      $(k,\ell) \leftarrow (H[i,j], G[i,j])$ # save input values for block $[i,j]$
      $(H[i,j], G[i,j]) \leftarrow (h+1, 0)$ # "eliminate" block $[i,j]$ by making it unusable
      if not GDD(H,G,h,g):
        $(H[i,j], G[i,j]) \leftarrow (k,\ell)$ # "restore" block $[i,j]$
  # Now, every block has been "eliminated," except those on an optimum drilling path.
  $k \leftarrow 1$:
  while $k \leqslant m$:
    select j_k such that $H[k,j_k] < h+1$—**break** out of the loop if this is not possible
    $k \leftarrow k+1$
  return $j_1, j_2, \ldots, j_{k-1}$

Correctness:

- The maximum value of g such that GDD(H,G,h,g) = True belongs in the range $[0,B]$ computed by the algorithm, because every drilling path goes through at most one block per level. Also, GDD$(H,G,h,g) \Rightarrow$ GDD$(H,G,h,g-1)$ and $\neg$GDD$(H,G,h,g) \Rightarrow \neg$GDD$(H,G,h,g+1)$, so binary search will correctly find the maximum value of g.
- GDD(H,G,h,g) is a loop invariant for the main loop. So at the end, the final values of H and G contain a drilling path with at least g gold. In addition, every block $[i,j]$ outside this drilling path will have $H[i,j] = h+1$ and $G[i,j] = 0$ because of the "elimination process" taking place during the loop. So an optimum drilling path is equal to the blocks $[i,j]$ where $H[i,j] < h+1$—exactly what the algorithm returns.

Runtime:

- Let b be the maximum number of bits needed to write down each of the values in matrices H and G, in addition to the value h (so the total size of the input is $s \leqslant b(mn+1)$).
- The size of B (in binary) is at most $b+m$. So performing binary search in the range $[0,B]$ makes $\mathcal{O}(\log B) = \mathcal{O}(s)$ many calls to GDD.
- The rest of the algorithm makes one call to GDD for each entry in the H and G matrices.
- In addition, the final loop takes time at most $\mathcal{O}(mn) = \mathcal{O}(s)$.
- The total running time of GDO is therefore $\mathcal{O}(sT(s))$, where $T(s)$ is the running time of GDD.

4. (a) PIR($x_1, x_2, \ldots, x_n$):

    $k \leftarrow 0$ # size of the subsequence
    $s \leftarrow 0$ # sum of the subsequence
    for $i \leftarrow 1, 2, \ldots, n$:
        if $s + x_i \leqslant B$:
            # Add x_i to the subsequence.
            $k \leftarrow k + 1$
            $i_k \leftarrow i$
            $s \leftarrow s + x_i$
    return $x_{i_1}, \ldots, x_{i_k}$

   (b) Let s^* denote the maximum possible sum for input $x_1, \ldots, x_n$.
   Either $x_1 \geqslant B/2$ or $x_1 < B/2$.

   - If $x_1 \geqslant B/2$, then the algorithm outputs a subsequence with sum $s \geqslant x_1 \geqslant B/2 \geqslant s^*/2$ (since $s^* \leqslant B$ by the problem definition).
   - If $x_1 < B/2$, then either the algorithm returns a subsequence with sum $s \geqslant B/2$ or the algorithm returns a subsequence with sum $s < B/2$.
     - If $s \geqslant B/2$ then, as in the very first case, $s \geqslant B/2 \geqslant s^*/2$.
     - If $s < B/2$ then $s = x_1 + x_2 + \cdots + x_n$ (the entire sequence is returned), so $s = s^* \geqslant s^*/2$.

   In all cases, $s \geqslant s^*/2$. By definition, the approximation ratio is at most 2.