Assignment # 1 (Due June 8, 11:59 pm)

Worth: 5%

1. [20 marks]
   Consider the problem of multiplying two n-bit integers x, y, and suppose that n is a multiple of 3 (if not, we add one or two zeroes on the left to make this true). Suppose that we split (the bits of) each integer into three equal parts (x into X_2, X_1, X_0 and y into Y_2, Y_1, Y_0).

   (a) [5 points]
   State the exact relationship between x and each of its three parts X_2, X_1, X_0 (*i.e.*, give an equation that involves x and X_2, X_1, X_0). Do the same for y and Y_2, Y_1, Y_0.

   (b) [12 points]
   Write a divide-and-conquer algorithm to multiply two integers x, y based on this three-way split. Justify that your algorithm produces the correct answer, *i.e.*, show that the output of your algorithm is equal to $x \cdot y$.
   For full marks, your algorithm must run in time $o(n^2)$, *i.e.*, strictly less than $\mathcal{O}(n^2)$. Justify that this is the case by computing the running time of your algorithm (you may use the Master Theorem as long as you state clearly how it applies to your algorithm).

   (c) [3 points]
   Is your algorithm faster or slower than the divide-and-conquer algorithm shown in class with a running time of $\Theta(n^{\log_2 3})$?

Solution. **(a)** We are considering the recursion step here, so we assume our numbers x and y have n bits where $n > 1$ is divisible by 3. Let X_2 be the $n/3$-bit number consisting of the bits of the leftmost third of x, let X_1 be the $n/3$-bit number consisting of the bits of the middle third of x, and let X_0 be the $n/3$-bit number consisting of the bits of the rightmost third of x. Define Y_2, Y_1, Y_0 similarly.

We have:

$$\begin{aligned} x &= X_2 \cdot 2^{2n/3} + X_1 \cdot 2^{n/3} + X_0 \\ y &= Y_2 \cdot 2^{2n/3} + Y_1 \cdot 2^{n/3} + Y_0. \end{aligned}$$

(b) The base case of $n = 1$ is obvious, so we assume our numbers have n bits where $n > 1$ is divisible by 3. We have the equations from part **(a)**. We wish to compute:

$$\begin{aligned} xy &= Z_4 \cdot 2^{4(n/3)} + Z_3 \cdot 2^{3(n/3)} + Z_2 \cdot 2^{2(n/3)} + Z_1 \cdot 2^{n/3} + Z_0, \text{ where} \\ Z_4 &= X_2 Y_2, \\ Z_3 &= X_2 Y_1 + X_1 Y_2, \\ Z_2 &= X_2 Y_0 + X_1 Y_1 + X_0 Y_2, \\ Z_1 &= X_1 Y_0 + X_0 Y_1, \\ Z_0 &= X_0 Y_0. \end{aligned}$$

Say that we can compute Z_4, Z_3, Z_2, Z_1, Z_0 using $c > 3$ recursive calls to multiplication of $n/3$ bit numbers, together with a constant number of additions. We can then compute xy using some shifts and additions, yielding a recurrence for our timing function T:

$$T(n) = cT(n/3) + O(n),$$

and a solution $T \in O(n^{\log_3 c})$.

It is easy to do this with $c = 7$ recursive calls, and not too hard to do it with $c = 6$ recursive calls. Here is one way:

$Z_0 \leftarrow X_0 \cdot Y_0$;
$Z_4 \leftarrow X_2 \cdot Y_2$;
$v_1 \leftarrow X_1 \cdot Y_1$;
$v_2 \leftarrow (X_0 + X_1) \cdot (Y_0 + Y_1)$;
$v_3 \leftarrow (X_1 + X_2) \cdot (Y_1 + Y_2)$;
$v_4 \leftarrow (X_0 + X_1 + X_2) \cdot (Y_0 + Y_1 + Y_2)$

$Z_1 \leftarrow v_2 - Z_0 - v_1$;
$Z_3 \leftarrow v_3 - Z_4 - v_1$;
$Z_2 \leftarrow v_4 - Z_0 - Z_1 - Z_3 - Z_4$

In fact, it can be done with $c = 5$ recursive calls!

Difficult Explanation: The trick is to view the problem as follows: we are given the coefficients of two degree 2 polynomials, and we want to compute the (five) coefficients of the product of the two polynomials. We do this by evaluating each of the two polynomials at five points (using 0 multiplications), then computing the value of the product polynomial at each of the five points (using 5 multiplications), and then doing *polynomial interpolation* (using 0 multiplications) to compute the desired coefficients. In this example, we will evaluate at the points $-2, -1, 0, 1, 2$.

Even if you didn't understand this explanation, you should still be able to check that the following works.

$u_2 \leftarrow (4X_2 + 2X_1 + X_0) \cdot (4Y_2 + 2Y_1 + Y_0)$;
$u_1 \leftarrow (X_2 + X_1 + X_0) \cdot (Y_2 + Y_1 + Y_0)$;
$u_0 \leftarrow X_0 \cdot Y_0$;
$u_{-1} \leftarrow (X_2 - X_1 + X_0) \cdot (Y_2 - Y_1 + Y_0)$;
$u_{-2} \leftarrow (4X_2 - 2X_1 + X_0) \cdot (4Y_2 - 2Y_1 + Y_0)$;

$Z_0 \leftarrow u_0$;
$Z_1 \leftarrow (1/12)(-u_2 + 8u_1 - 8u_{-1} + u_{-2})$;
$Z_2 \leftarrow (1/24)(-u_2 + 16u_1 - 30u_0 + 16u_{-1} - u_{-2})$;
$Z_3 \leftarrow (1/12)(u_2 - 2u_1 + 2u_{-1} - u_{-2})$
$Z_4 \leftarrow (1/24)(u_2 - 4u_1 + 6u_0 - 4u_{-1} + u_{-2})$

(c) If you have $c = 6$, then you get $T \in O(n^{\log_3 6})$. $\log_3 6$ is about 1.63 and $\log_2 3$ is about 1.58, so we see that this yields an algorithm that is worse than the one in the text.

However, if you have $c = 5$, then you get $T \in O(n^{\log_3 5})$. $\log_3 5$ is about 1.46 so we see that this yields an algorithm that is (asymptotically) better than the one in the text. □

2. [20 marks]
Consider a variant on the problem of Interval Scheduling where instead of wanting to schedule as many jobs as we can on one processor, we now want to schedule ALL of the jobs on as few processors as possible.

The input to the problem is $(s_1, f_1), (s_2, f_2), \cdots, (s_n, f_n)$, where $n \geq 1$ and all $s_i < f_i$ are nonnegative integers. The integers s_i and f_i represent the start and finish times, respectively, of job i.

A *schedule* specifies for each job i a positive integer $P(i)$ (the "processor number" for job i). It must be the case that if $i \neq j$ and $P(i) = P(j)$, then jobs i and j do not overlap. We wish to find a schedule that uses as few processors as possible, *i.e.*, such that $\max\{P(1), P(2), \cdots, P(n)\}$ is minimal.

(a) [5 marks]
Design an algorithm (write a pseudocode) to solve the above problem in time $o(n^2)$, *i.e.*, strictly less than $\mathcal{O}(n^2)$.

(b) [10 marks]
Prove that the above algorithm is guaranteed to compute a schedule that uses the minimum number of processors.

(c) [5 marks]
Briefly describe an efficient implementation of the algorithm, making it clear what data structures you are using. Express the running time of your implementation as a function of n (the number of jobs), using appropriate asymptotic notation.

Solution. **(a)** Consider the following greedy algorithm: Sort the jobs by nondecreasing start time, then for each job, schedule it on some existing processor if that is possible (note that if more than one processor can be used, the algorithm doesn't specify which one to choose); if no processor is free, create a new processor and schedule the job on the new processor. The algorithm uses variable m to store the number of processors currently being used. For each processor j, $E[j]$ is the last finish time of the jobs currently scheduled on processor j.

```
sort jobs so that $s_1 \le s_2 \le \cdots \le s_n$
$m := 1$
$P[1] := 1$
$E[1] := f_1$
for $i := 2$ to $n$:
    if there is a processor number $j$ such that $1 \le j \le m$ and $E[j] \le s_i$:
        $j :=$ some processor number such that $E[j] \le s_i$
        $P[i] := j$
        $E[j] := f_i$
    else:
        $m := m + 1$
        $P[i] := m$
        $E[m] := f_i$
```

(b) We will first prove this using the "standard" method described in class. We will then show a simpler proof.

We will prove by induction on i, $0 \le i \le n$, that the partial schedule computed by the algorithm $P(1), P(2), \ldots, P(i)$ has the property that there exists an optimal schedule OPT such that $OPT(1) = P(1), OPT(2) = P(2), \ldots, OPT(i) = P(i)$.
(This implies that the algorithm produces an optimal schedule.)

Base Case: $i = 0$. This is trivially true, since any optimal schedule OPT will work.

Induction Step: Let $0 \le i < n$, and let OPT be an optimal schedule such that $OPT(1) = P(1)$, $OPT(2) = P(2)$, …, $OPT(i) = P(i)$. Also, assume that $\max\{P(1), P(2), \ldots, P(i)\} = m$.

CASE 1: $P(i+1) = m+1$.
In this case, for every processor $j \le m$, job $i+1$ overlaps with a job (from the first i jobs) on processor j, so we must also have $OPT(i+1) > m$. If $OPT(i+1) = m+1$, we are done. So, assume that $OPT(i+1) = k > m+1$. Define OPT' to be the schedule obtained from OPT by interchanging all the

jobs on processor k with all the jobs on processor $m+1$. OPT' is an optimal schedule with the desired property.

CASE 2: $P(i+1) = k \leq m$.
If $OPT(i+1) = k$, we are done. So, assume that $OPT(i+1) = k' \neq k$. This means that job $i+1$ doesn't overlap with any of the jobs $1, 2, \ldots, i$ that are scheduled on processors k or k'. Since the jobs are sorted in order of nondecreasing start times, this means that after jobs $1, 2, \ldots i$ are scheduled, both processors k and k' are free starting at time s_{i+1}. Let OPT' be the schedule obtained from OPT by interchanging all the jobs on processor k that start at or after s_{i+1} with all the jobs on processor k' that start at or after s_{i+1}. OPT' is an optimal schedule with the desired property.

Simpler Proof:
Say that the maximum processor number used by the algorithm is m. Consider the first time the algorithm puts a job, say i, on processor m. This means that at this time the interval (s_i, f_i) overlaps with a job on each of the processors $1, 2, \ldots, m-1$. Because the jobs are sorted in order of nondecreasing start times, there exists at this time, on each processor $\in \{1, 2, \ldots, m-1\}$, a job that starts at or before s_i that overlaps with (s_i, f_i). Let S be the set of these jobs, together with job i. Every two jobs from S overlap with each other, and there are m jobs in S. This means that no schedule can use fewer than m processors. So the schedule produced by the algorithm is optimal.

(This is an example of a proof by lower bounds.)

(c) Sorting the jobs by their start times can be done in time $\Theta(n \log n)$.
For the rest of the algorithm, we will use a Priority Queue Q. For each processor j that is in use, we will have $(j, E(j))$ in Q, with key $E(j)$. The operations involve examining the min element from Q, removing the min element from Q, and adding an element to Q, each of which takes $\Theta(\log n)$ time. There are n such operations in total, for a time of $\Theta(n \log n)$.
So, the total time taken by the algorithm is $\Theta(n \log n)$. □

3. [20 marks]
Here is another variant on the problem of Interval Scheduling.

Suppose we now have *two* processors, and we want to schedule as many jobs as we can. As before, the input is $(s_1, f_1), (s_2, f_2), \cdots, (s_n, f_n)$ where $n \geq 1$ and all $s_i < f_i$ are nonnegative integers.

A *schedule* is now defined as a pair of sets (A_1, A_2), the intuition being that A_i is the set of jobs scheduled on processor i. A schedule must satisfy the obvious constraints: $A_1 \subseteq \{1, 2, \ldots, n\}$, $A_2 \subseteq \{1, 2, \ldots, n\}$, $A_1 \cap A_2 = \emptyset$, and for all $i \neq j$ such that $i, j \in A_1$ or $i, j \in A_2$, jobs i and j do not overlap.

(a) [5 marks]
Design an algorithm (write a pseudocode) to solve the above problem in time $o(n^2)$, *i.e.*, strictly less than $\mathcal{O}(n^2)$.

(b) [10 marks]
Prove that the above algorithm is guaranteed to compute an optimal schedule.

(c) [5 marks]
Briefly describe an efficient implementation of the algorithm, making it clear what data structures you are using. Express the running time of your implementation as a function of n (the number of jobs), using appropriate asymptotic notation.

Solution. **(a)** A natural greedy algorithm for this variation is as follows. First, sort the jobs in order of nondecreasing finish times. Then, handle the jobs one at a time, scheduling each job if possible. If a job can be scheduled on either processor, pick the processor where it will cause the smallest "gap". This algorithm is described in detail below, where variables E_1 and E_2 are used to keep track of the largest finish time of jobs in A_1 and A_2, respectively.

```
sort jobs so that $f_1 \le f_2 \le \cdots \le f_n$
$A_1 = \varnothing$
$A_2 = \varnothing$
$E_1 := 0$
$E_2 := 0$
for $i := 1$ to $n$:
    if $E_2 \le E_1 \le s_i$ or $E_1 \le s < E_2$:
        $A_1 := A_1 \cup \{s_i\}$
        $E_1 := f_i$
    else if $E_2 \le s_i$:
        $A_2 := A_2 \cup \{s_i\}$
        $E_2 := f_i$
```

(b) Let us denote the schedule created by the algorithm after examining the first i jobs by (A_1^i, A_2^i); let the values of E_1 and E_2 be E_1^i and E_2^i. We will prove by induction on i, $0 \le i \le n$, that there is an optimal schedule (OPT_1, OPT_2) such that $A_1^i = OPT_1 \cap \{1,2,\ldots,i\}$ and $A_2^i = OPT_2 \cap \{1,2,\ldots,i\}$. (This implies that the algorithm produces an optimal schedule.)

Base Case: $i = 0$. This is trivially true, since any optimal schedule will work.

Induction Step: Let $0 \le i < n$, and let (OPT_1, OPT_2) be an optimal schedule such that $A_1^i = OPT_1 \cap \{1,2,\ldots,i\}$ and $A_2^i = OPT_2 \cap \{1,2,\ldots,i\}$.

CASE 1: The algorithm does not add job $i+1$ to either A_1^i or A_2^i.
Then job $i+1$ overlaps with a member of A_1^i and with a member of A_2^i, so $i+1$ cannot be in either OPT_1 or OPT_2. So $A_1^{i+1} = OPT_1 \cap \{1,2,\ldots,i+1\}$ and $A_2^{i+1} = OPT_2 \cap \{1,2,\ldots,i+1\}$.

CASE 2: $A_1^{i+1} = A_1^i \cup \{i+1\}$.
If $i+1 \in OPT_1$ we are done, so assume $i+1 \notin OPT_1$. We have $E_1^i \le s_{i+1}$.

Subcase 2A: $i+1 \in OPT_2$. (This is the most interesting case.)
The algorithm could have put job $i+1$ on processor 2 (since OPT did), and since it didn't, we must have $E_2^i \le E_1^i \le s_{i+1}$.
We can write $OPT_1 = A_1^i \cup B$ and $OPT_2 = A_2^i \cup C$ where $B, C \subseteq \{i+1,\ldots,n\}$ and $i+1 \in C$. Since the jobs were sorted according to finish time and $i+1 \in C$, every job in B starts at time $\ge E_1^i$ and every job in C starts at time $\ge s_{i+1}$. So every job in B starts at time $\ge E_2^i$ and every job in C starts at time $\ge E_1^i$. This means that we can interchange B and C, so that $(A_1^i \cup C, A_2^i \cup B)$ is a schedule; since it has the same size as $(A_1^i \cup B, A_2^i \cup C)$, it is an optimal schedule. Call this schedule (OPT_1', OPT_2'). Then, we have $A_1^{i+1} = (A_1^i \cup C) \cap \{1,2,\ldots,i+1\} = OPT_1' \cap \{1,2,\ldots,i+1\}$, and $A_2^{i+1} = (A_2^i \cup B) \cap \{1,2,\ldots,i+1\} = OPT_2' \cap \{1,2,\ldots,i+1\}$.

Subcase 2B: $i+1 \notin OPT_1$ and $i+1 \notin OPT_2$.
As in the one processor proof, since the jobs are sorted according to nondecreasing finish times, there is only one job $j \in OPT_1$ that overlaps job $i+1$, and $j > i+1$. Let $OPT_1' = OPT_1 \cup \{i+1\} - \{j\}$. Then (OPT_1', OPT_2) is an optimal schedule with the desired properties.

CASE 3: $A_2^{i+1} = A_2^i \cup \{i+1\}$. This is similar to Case 2.

(c) Sorting the jobs by their finish times can be done in time $\Theta(n \log n)$.
For the rest of the algorithm, there is only one loop that runs n times, and in each iteration takes $\Theta(1)$ time, for a total of $\Theta(n)$ time.
So, the total time taken by the algorithm is $\Theta(n \log n)$. □

4. [20 marks]
Consider the problem of making change, given a finite number of coins with various values. Formally:

Input: A list of positive integer coin values $c_1, c_2, \cdots, c_m$ (with repeated values allowed) and a positive integer amount A.

Output: A selection of coins $\{i_1, i_2, \dots, i_k\} \subseteq \{1, 2, \dots, m\}$ such that $c_{i_1} + c_{i_2} + \cdots + c_{i_k} = A$ and k is as small as possible. If it is impossible to make change for amount A exactly, then the output should be the empty set $\emptyset$.

(a) [5 points]
Describe a natural greedy strategy you could use to try and solve this problem, and show that your strategy does not work.
(The point of this question is **not** to try and come up with a really clever greedy strategy — rather, we simply want you to show why the "obvious" strategy fails to work.)

(b) [10 points]
Give a detailed dynamic programming algorithm to solve this problem. Follow the steps outlined in class, and include a brief (but convincing) argument that your algorithm is correct.

(c) [5 points]
What is the worst-case running time of your algorithm? Justify briefly.

Solution. **(a) Greedy strategy:** Repeatedly pick the largest coin remaining.

Counter-example: Consider input $A = 30$, $c_1 = 25$, $c_2 = 10$, $c_3 = 10$ and $c_4 = 10$. The greedy algorithm would pick $c_1 = 25$ and be unable to finish making change, even though $\{c_2, c_3, c_4\}$ is a solution.

(b) Step 1: Describe the recursive structure of sub-problems.
For every optimal solution $i_1 < i_2 < \cdots < i_k$, either $i_k = m$ or $i_k < m$.
If $i_k = m$, then $\{i_1, i_2, \dots, i_{k-1}\}$ must be an optimal solution for input $A - c_m, c_1, \dots, c_{m-1}$ (if there were a better solution for input $A - c_m, c_1, \dots, c_{m-1}$, we could simply add c_m to it and get a better overall solution).
If $i_k < m$, then $\{i_1, i_2, \dots, i_k\}$ must be an optimal solution for input $A, c_1, \dots, c_{m-1}$ (trivially).

Step 2: Define an array that stores optimal values for arbitrary sub-problems.
Define $N[k, a]$ to be the minimum number of coins required to make change for amount a using coins $c_1, \dots, c_k$, for $0 \le a \le A, 0 \le k \le m$. (Let $N[k, a] = \infty$ when the problem has no solution for input $a, c_1, \dots, c_k$.)

Step 3: Give a recurrence relation for the array values. (We include justifications for each case of the recurrence.)
$N[k, 0] = 0$ for $0 \le k \le m$ (no coin is necessary to make change for amount 0).
$N[0, a] = \infty$ for $1 \le a \le A$ (it is impossible to make change for a positive amount without using any coins).

$N[k,a] = N[k-1,a]$ if $c_k > a$, for $1 \le k \le m, 1 \le a \le A$ (coin c_k cannot be used to make change for amount a if $c_k > a$).
$N[k,a] = \min\{N[k-1,a], 1+N[k-1,a-c_k]\}$ if $c_k \le a$, for $1 \le k \le m, 1 \le a \le A$ (any optimal solution either does not use c_k, or it does).

Step 4: Write a bottom-up algorithm to compute the array values, following the recurrence.

```
# Simply fill in the array values, following the recurrence.
N[0, 0] := 0
for a := 1,2,...,A:  N[0, a] := oo
for k := 1,2,...,m:
    N[k, 0] := 0
    for a := 1,2,...,A:
        N[k, a] := N[k-1, a]
        if c_k <= a and 1 + N[k - 1, a - c_k] < N[k, a]:
            N[k, a] := 1 + N[k - 1, a - c_k]
```

Step 5: Use the computed values to reconstruct an optimal solution.

```
# Idea: for every k, a, use coin c_k iff N[k, a] != N[k - 1, a].
if N[m, A] = oo:  return {}
S = {}
a := A
for k := m, m - 1,...,1:
    if N[k, a]  !=  N[k - 1, a]:
        S := S u {k}
        a := a - c_k
return S
```

(c) The worst-case runtime of the entire algorithm is $\Theta(mA)$, for filling in the values of $N[k,a]$ (reconstructing the solution takes only time $\Theta(m)$).

□

5. [20 marks]
During the renovations at Union Station, the work crews excavating under Front Street found veins of pure gold ore running through the rock! They cannot dig up the entire area just to extract all the gold: in addition to the disruption, it would be too expensive. Instead, they have a special drill that they can use to carve a single path into the rock and extract all the gold found on that path. Each crew member gets to use the drill once and keep the gold extracted during their use. You have the good luck of having an uncle who is part of this crew. What's more, your uncle knows that you are studying computer science and has asked for your help, in exchange for a share of his gold!

The drill works as follows: starting from any point on the surface, the drill processes a block of rock 10cm × 10cm × 10cm, then moves on to another block 10cm below the surface and connected with the starting block either directly or by a face, edge, or corner, and so on, moving down by 10cm at each "step". The drill has two limitations: it has a maximum depth it can reach and an initial hardness that gets used up as it works, depending on the hardness of the rock being processed; once the drill is all used up, it is done even if it has not reached its maximum depth.

The good news is that you have lots of information to help you choose a path for drilling: a detailed geological survey showing the hardness and estimated amount of gold for each 10cm × 10cm × 10cm block of rock in the area. To simplify the notation, in this homework, you will solve a two-dimensional version of the problem defined as follows.

Input: A positive integer d (the initial *drill hardness*) and two $[m \times n]$ matrices H, G containing non-negative integers. For all $i \in \{1,\dots,m\}, j \in \{1,\dots,n\}$, $H[i,j]$ is the *hardness* and $G[i,j]$ is the *gold content* of the block of rock at location i,j (with $i = 1$ corresponding to the surface and $i = m$ corresponding to the maximum depth of the drill).

There is one constraint on the values of each matrix: $H[i,j] = 0 \implies G[i,j] = 0$ (blocks with hardness 0 represent blocks that have been drilled already and contain no more gold).

Figure 1 below shows the general form of the input. Figure 2 shows a sample input.

Output: A drilling path $j_1, j_2, \dots, j_\ell$ for some $\ell \le m$ such that:

- $\boxed{1 \le j_k \le n \textbf{ for } k = 1,2,\dots,\ell \textbf{ (each coordinate on the path is valid);}}$
- $j_{k-1} - 1 \le j_k \le j_{k-1} + 1$ for $k = 2,\dots,\ell$ (each block is underneath the one just above, either directly or diagonally);
- $H[1,j_1] + H[2,j_2] + \cdots + H[\ell,j_\ell] \le d$ (the total hardness of all the blocks on the path is no more than the initial drill hardness);
- $G[1,j_1] + G[2,j_2] + \cdots + G[\ell,j_\ell]$ is maximum (the path collects the maximum amount of gold possible).

Follow the dynamic programming paradigm given in class (the "five step" process) to give a detailed solution to this problem. Make sure to keep a clear distinction between each of the steps, and to explain what you are doing and why at each step — you will **not** get full marks if your answer contains only equations or algorithms with no explanation or justification.

$H[1,1], G[1,1]$	$H[1,2], G[1,2]$	$\cdots$	$H[1,n], G[1,n]$
$H[2,1], G[2,1]$	$H[2,2], G[2,2]$	$\cdots$	$H[2,n], G[2,n]$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
$H[m,1], G[m,1]$	$H[m,2], G[m,2]$	$\cdots$	$H[m,n], G[m,n]$

Figure 1: General form of the input matrix.

2,2	**2,7**	0,0	2,3	1,8
2,2	**0,0**	1,2	2,0	1,1
1,4	1,1	**2,6**	2,1	3,8

Figure 2: Sample input with optimum path shown in **bold** (for $d = 4$).

Solution. **Step 1:** *Recursive Structure.*

Suppose $j_1, j_2, \dots, j_\ell$ is an optimum drilling path. Then $j_2, j_3, \dots, j_\ell$ is an optimum drilling path starting at one of the coordinates $(2, j_1 - 1), (2, j_1), (2, j_1 + 1)$ and with maximum drill hardness $d - H[1, j_1]$; if there were a better path starting from one of those coordinates and with the same maximum hardness, we could follow it after block $(1, j_1)$ to get more gold overall.

Step 2: *Array Definition.*

Let $M[i,j,h]$ denote the maximum amount of gold that can be drilled starting from coordinates (i,j) with drill hardness h, for $1 \le i \le m+1$, $0 \le j \le n+1$ and $0 \le h \le d$.

Step 3: *Recurrence Relation.*

For $0 \le h \le d$:

- $M[i,0,h] = M[i,n+1,h] = -\infty$ for $1 \le i \le m+1$ (regions outside of the geological survey cannot be drilled — setting $M = -\infty$ ensures that the drilling path does not stray outside of the surveyed region);
- $M[m+1,j,h] = 0$ for $1 \le j \le n$ (no gold is accessible below depth m);
- for $1 \le i \le m$ and $1 \le j \le n$,
  - $M[i,j,h] = 0$ if $h < H[i,j]$ (not enough hardness to drill block (i,j)),
  - $M[i,j,h] = G[i,j] + \max\left\{ M\big[i+1, j-1, h-H[i,j]\big], M\big[i+1, j, h-H[i,j]\big], M\big[i+1, j+1, h-H[i,j]\big] \right\}$ if $h \ge H[i,j]$, (get the gold from block (i,j) and do the best possible with the remaining drill hardness, starting one block below (i,j)).

Step 4: *Iterative Algorithm.*

```
for $h := 0, 1, \ldots, d$:
    # Fill in values for depth $m+1$.
    $M[m+1, 0, h] := -\infty$
    $M[m+1, n+1, h] := -\infty$
    for $j := 1, 2, \ldots, n$:
        $M[m+1, j, h] := 0$
    # Compute values from deepest to shallowest level.
    for $i := m, m-1, \ldots, 1$:
        $M[i, 0, h] := -\infty$
        $M[i, n+1, h] := -\infty$
        for $j := 1, 2, \ldots, n$:
            if $h < H[i,j]$:
                $M[i,j,h] := 0$
            else:
                $M[i,j,h] := G[i,j] + \max\{ M[i+1, j-1, h-H[i,j]],$
                                          $M[i+1, j, h-H[i,j]],$
                                          $M[i+1, j+1, h-H[i,j]] \}$
```

Runtime: $\Theta(dmn)$. This is pseudopolynomial time because of d.

Step 5: *Solution Reconstruction.*

```
$h := d$    # current hardness
$\ell := 1$    # current depth
Find $j_\ell \in \{1, \ldots, n\}$ that maximizes $M[\ell, j_\ell, h]$.    # start of drilling path
while $M[\ell, j_\ell, h] > 0$:
    # It's possible to get more gold starting from current coordinates $(\ell, j_\ell)$ with drill
    # hardness $h$: keep block $(\ell, j_\ell)$ on the drilling path and figure out the next block.
    $h := h - H[\ell, j_\ell]$
    $\ell := \ell + 1$
    Find $j_\ell \in \{j_{\ell-1} - 1, j_{\ell-1}, j_{\ell-1} + 1\}$ that maximizes $M[\ell, j_\ell, h]$.
return $j_1, j_2, \ldots, j_{\ell-1}$
```

Additional runtime: $\Theta(n+m)$ ($\Theta(n)$ to find $j_1 \in \{1, \ldots, n\}$ and $\Theta(m)$ to find each successive j_ℓ).

□