This document gives model solutions to the assignment problems. Note that alternative solutions may exist.

Question 1

Given a strictly increasing sequence A whose entries are positive integers $[a_1, \ldots, a_n]$, a sub-sequence of A is defined by removing zero or more elements from A while preserving the original order. For example, $[1, 13, 21]$ and $[1, 2, 3, 13, 18, 21, 71]$ are both sub-sequences of $[1, 2, 3, 13, 18, 21, 71]$.

A sequence $x_1, x_2, ..., x_m$ is called *beautiful* if:

- $m \geq 3$, that is, the number of elements in the sequence is greater or equal to 3.
- $3x_i + 5x_{i+1} = x_{i+2}$ for all $i \leq n - 2$.

In the example above, the beautiful sub-sequences are:

- $[1, 2, 13]$,
- $[1, 3, 18]$,
- $[2, 3, 21]$,
- $[2, 13, 71]$ and
- $[1, 2, 13, 71]$.

1.1 [5 marks] Suppose $[\ldots, a_j, a_k]$ is a beautiful sub-sequence of A. Given the indices j and k of the last two entries of the sub-sequence, design an algorithm which runs in $O(\log n)$ time and computes the index in A of the third last entry of the sub-sequence.

Your algorithm should find the index i of the entry a_i which precedes a_j and a_k in the sequence, assuming that such an i exists.

This can be solved with binary search. Supposed the index in A of the third last element of the beautiful sub-sequence is called i. We know from the definition of a beautiful sub-sequence that $a_i = \frac{a_k - 5a_j}{3}$. Hence, to find i, we can binary search the array A between indices 1 and j to find the index of the element with value $\frac{a_k - 5a_j}{3}$.

Correctness follows directly from directness of binary search. If the element exists in A, it will be found by binary search. Since A is strictly increasing, the result will also be unique. The time complexity is $O(\log n)$ as we are performing a binary search on an array of at most n elements.

1.2 [12 marks] Design an algorithm which runs in $O(n^2 \log n)$ time and computes the length of the longest beautiful sub-sequence of A.

We believe this problem can be solved in $O(n^2)$ time. A solution satisfying this constraint will earn one bonus mark for the course.

An $O(n^2 \log n)$ solution.

Subproblems:
Let $\text{opt}(i, j)$ be the length of the longest beautiful sub-sequence where the last two elements have indexes in A equal to i and j, defined for all $1 \leq i < j \leq n$.

Recurrence:

$$\text{opt}(j,k) = \begin{cases} 1 + \text{opt}(i,j) & \text{there exists } 0 < i < j \text{ such that } 3a_i + 5a_j = a_k \\ 2 & \text{otherwise} \end{cases}$$

Checking for a valid i can be done by binary searching in $A[1..j-1]$ for $\frac{a_k - 5a_j}{3}$ as described in part 1.

Base Case:
$\text{opt}(1,k) = 2$ for all $1 < k \leq n$.

Order of Computation:
Before solving $\text{opt}(j,k)$, we must ensure $\text{opt}(i,j)$ has been solved for all $i < j$. Since $j < k$ by definition, this can be guaranteed by solving subproblems in order of increasing k (in any order of j).

Final Solution:
Let $M = \max_{1 \leq i < j \leq n} \text{opt}(i,j)$ by the largest value of $\text{opt}(i,j)$. The final solution is M if $M > 2$, and 0 otherwise.

Justification:
The base case correctness is clear, since when $j = 1$ there cannot be a beautiful sub-sequence ending with elements j and k since no elements come before j. The recursion correctness follows from the observation that the longest beautiful sub-sequence ending with elements j, k must be precisely one longer than the longest beautiful sub-sequence ending with elements i, j for the single value i that satisfies the required equation (if it exists).

The final solution is simply the longest beautiful sub-sequence ending in any two elements. In the event that there are no beautiful sub-sequences, then we will simply finish with $\text{opt}(j,k) = 2$ for all j, k, and this case is also handled correctly by the final solution.

Time Complexity:
There are $O(n^2)$ subproblems and each subproblem requires a binary search of complexity $O(\log n)$, so the total time complexity will be $O(n^2 \log n)$ as was required.

An $O(n^2)$ solution.

The above solution can be improved to $O(n^2)$ by simply changing the method in which the subproblems are computed.

Fix a particular value j. Then maintain two pointers i and k, where initially $i = 1$ and $k = j + 1$. Then proceed as follows:

- if $3a_i + 5a_j < a_k$, then advance i (no further than j), as the value at index i is too small to precede a_j and anything larger than or equal to a_k in a beautiful subsequence
- if $3a_i + 5a_j > a_k$, then advance k (no further than n), as the value at index k is too small to follow a_j and anything larger than or equal to a_i in a beautiful subsequence
- if $3a_i + 5a_j = a_k$, then update $\text{opt}(j,k)$ to $opt(i,j) + 1$, and advance both i and k.

This takes $O(n)$ for each j, for a total of $O(n^2)$.

1.3 **[3 marks]** Design an algorithm which runs in $O(n \log n)$ additional time and lists the entries of the longest beautiful sub-sequence of A.

'Additional' here means after the execution of your algorithm for 1.2.

If there are two or more equal longest beautiful sub-sequences, your algorithm should produce any one of them.

Assume that the algorithm of 1.2 has completed and we know the values of j, k that maximised $\text{opt}(j, k)$. Starting with the k, j, we can backtrack through the array A by using the binary search explained in part 1. This is summarised by the following steps:

1) Append a_k and a_j to the output.
2) Let $r, t = a_k, a_j$
3) Binary search in A for $\frac{r-5t}{3}$. If it can be found, then append $\frac{r-5t}{3}$ to the output, let $r = t$ and $t = \frac{r-5t}{3}$ and repeat this step.
4) Reverse the output to get the beautiful sub-sequence in ascending order.

Since $\text{opt}(j, k) < n$, this method takes a total of $O(n \log n)$ time, as we performed a binary search to back track for each element in the sub-sequence.

An $O(n)$ solution.

Assume that the algorithm of 1.2 has completed and we know the values of j, k that maximised $\text{opt}(j, k)$, and the length of the longest beautiful sub-sequence is ℓ. We will call the sub-sequence $x_1, ..., x_\ell$. Starting with the k, j, we can backtrack through the array A by solving the equation. This is summarised by the following steps:

1) Let $x_\ell = a_k$ and $x_{\ell-1} = a_j$.
2) For each r from $\ell - 2$ to 1, $x_r = \frac{x_{r+2}-5x_{r+1}}{3}$.

This takes a total of $O(\ell) = O(n)$ time, since each step in backtracking can be done in constant time.

Question 2

You are in a warehouse represented as a grid with m rows and n columns, where $m, n \geq 2$. You are initially located at the top-left cell $(1, 1)$, and there is an exit at the bottom-right cell (m, n). There are certain cells that contain boxes which you can not move through. The grid is given as a 2D array $B[1..m][1..n]$ where $B[i][j]$ is TRUE if cell (i, j) contains a box and FALSE otherwise.

2.1 [6 marks] Occupational health and safety regulations specify that it must be possible to reach the exit from your starting point by making only two kinds of moves: down one cell or right one cell. Design an algorithm that runs in $O(mn)$ time and determines whether the warehouse layout meets this requirement.

Note: there is both a DP and a non-DP approach to this question. Both are eligible for full marks in this part, but the DP approach naturally lends itself to be extended to Question 2.2. If you choose the non-DP approach here, you will likely need to put in more work to get full marks in Question 2.2.

DP approach:

Subproblem

Let $r(i, j)$ be whether cell (i, j) can be reached from the top left cell $(1, 1)$.

Recurrence

$$r(i,j) = \begin{cases} \textsc{False} & \text{if } B[i][j] = \textsc{True}, \\ r(i-1,1) & \text{if } j = 1, \\ r(1,j-1) & \text{if } i = 1, \\ r(i,j-1) \text{ OR } r(i-1,j) & \text{otherwise.} \end{cases}$$

Any cell containing a box can of course not be reached. Otherwise, if a cell is on the left edge of the warehouse ($j = 1$), then it can only be reached from the cell directly above it $(i-1,1)$, so it is reachable if and only if that cell is. The case where $i = 1$ is similar.

Otherwise, the cell (i,j) can potentially be reached from either the cell above or the cell to the left, both of which are within the warehouse. The cell is reachable if and only if either of these are reachable, as we can move from either of these directly to (i,j).

Base case

The start cell $(1,1)$ is reachable if it is not a box, so $r(1,1) = \text{NOT } B[1][1]$.

Computation

The answer is whether we can reach (m,n) from $(1,1)$, which is given by $r(m,n)$.

Each subproblem $r(i,j)$ depends on $r(i,j-1)$ and $r(j,i-1)$, so we can solve the subproblems in lexicographic order (increasing order of i then j). There are mn subproblems, each solved in constant time, so the algorithm runs in $O(mn)$ time.

Non-DP approach:

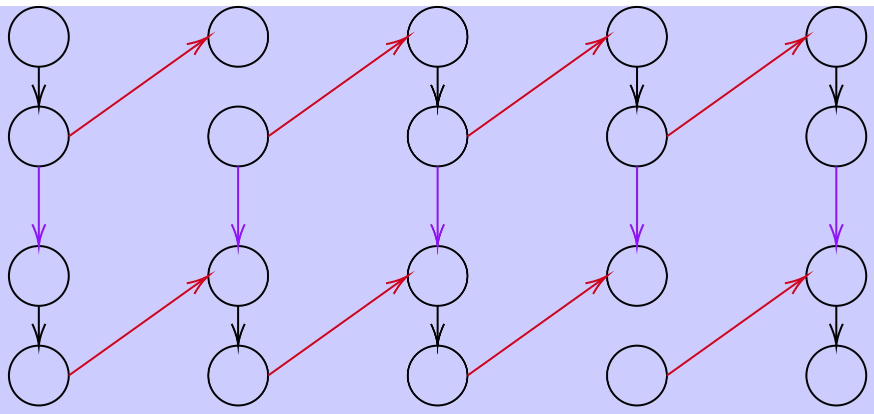
We construct a directed graph with:

- Two vertices $a_{i,j}$ and $b_{i,j}$ for each cell (i,j) of the warehouse,
- An edge $a_{i,j} \to b_{i,j}$ if $B[i][j] = \textsc{False}$,
- An edge $b_{i,j} \to a_{i+1,j}$ if cell $(i+1,j)$ exists,
- An edge $b_{i,j} \to a_{i,j+1}$ if cell $(i,j+1)$ exists.

For example, given the warehouse layout

	B			
			B	

we would construct the following graph.



The red edges represent moving right ($b_{i,j} \to a_{i,j+1}$), the purple edges represent moving down ($b_{i,j} \to a_{i+1,j}$), and the black edges represent passing through a cell. Note that although we can reach the exit if we move upwards from $(2,3)$ to $(1,3)$, the directed edges ensure that such a path does not exist in the graph.

The warehouse meets the requirement if and only if there is a path from $a_{1,1}$ to $b_{m,n}$, which we determine using a breadth-first search.

If there is some path through this graph, then it can be translated into a valid path through the warehouse. As all the edges are directed, all moves will be either right or down, and as there is no edge $a_{i,j} \to b_{i,j}$ if (i,j) contains a box, no moves will go through a square containing a box. Hence, if the warehouse meets the safety requirement, the graph contains a path from $a_{1,1}$ to $b_{m,n}$.

Conversely, if there is no valid path through the warehouse, the graph contains no path from $a_{1,1}$ to $b_{m,n}$. We show this by contradiction - suppose that the graph contains such a path, but there is no valid path through the warehouse. Then, the path in the graph must go through a vertex representing a cell that has a box, or it must contain an upward or left move. The former is not possible, as to move through a cell you must go from $a_{i,j}$ to $b_{i,j}$, but there is no such edge in the graph is cell (i,j) contains a box. The latter is also impossible, as all edges in the graph are either from $a_{i,j}$ to $b_{i,j}$, or a directed edge to a vertex representing cell $(i+1,j)$ or $(i,j+1)$. Hence, if the warehouse does not meet the safety requirement, the graph contains no path from $a_{1,1}$ to $b_{m,n}$,

The graph contains $2mn$ vertices, and at most $3mn$ edges, so it can be constructed in $O(mn)$ time. Performing a BFS with an adjacency list representation then takes $O(V+E) = O(2mn+3mn) = O(mn)$ time.

2.2 [3 marks] Unfortunately, some warehouse layouts do not meet this requirement. You have been asked to remove some boxes to ensure that the requirement is met, but wish to remove as few as possible to make efficient use of warehouse space. Design an algorithm that runs in $O(mn)$ time and determines the smallest number of boxes that must be removed to meet the requirement.

Subproblem

Let $\text{opt}(i,j)$ be the minimum number of boxes that must be removed to ensure that (i,j) can be reached from the top left cell $(1,1)$.

Recurrence

We can reach (i, j) from either $(i-1, j)$ or $(i, j-1)$, of which we should choose whichever requires removing the least boxes to get to. We then have to remove the box at (i, j) if it exists, adding an extra one to the count. This gives the recurrence:

$$\operatorname{opt}(i,j) = \min(\operatorname{opt}(i,j-1), \operatorname{opt}(i-1,j)) + \begin{cases} 1 & \text{if } B[i][j], \\ 0 & \text{otherwise.} \end{cases}$$

Base case

If cell $(1, 1)$ contains a box, then we must remove it, so $\operatorname{opt}(1, 1) = 1$. Otherwise, cell $(1, 1)$ is already reachable, so $\operatorname{opt}(1, 1) = 0$.

If $i = 0$ or $j = 0$, the cell (i, j) is outside the warehouse and can never be reached no matter how many boxes are removed, so $\operatorname{opt}(i, j) = \infty$.

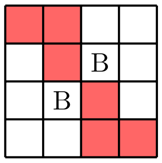
Computation

The answer is the number of boxes that must be removed to reach cell (m, n), given by $\operatorname{opt}(m, n)$.

As in the previous part, the subproblems can be solved in lexicographic order. Each of the mn subproblems are solved in $O(1)$, so the algorithm runs in $O(mn)$.

The fire alarm has gone off, so you have to make your escape! Fortunately, this warehouse passed the safety inspection, so you know there is a path to the exit that only involves moving down or right one cell at a time. However, you also want to ensure you can make a speedy escape, so you must take **exactly one** shortcut by moving diagonally: one cell down and right at the same time.

For example, the red path through this warehouse includes a shortcut from $(2, 2)$ to $(3, 3)$.



2.3 **[2 marks]** Show that any warehouse passing the safety requirement from 2.1 has a path from $(1, 1)$ to (m, n) that includes a shortcut.

Since the warehouse has dimensions of at least 2×2, we must make at least one move right and one move down. We can find some point where two different moves are conducted consecutively and replace them with a shortcut.

2.4 **[9 marks]** Each cell of the warehouse that does not contain a box has a particular hazard rating $H[i][j]$, and you want to minimise the sum of the hazard ratings on your path to the exit.

For example, the red path through this warehouse layout has a total hazard rating of 13, takes exactly one shortcut, and is the minimal path for this particular warehouse.

2	**2**	6	1
3	**4**	B	1
2	B	**2**	2
1	4	**2**	**1**

Design an algorithm that runs in $O(mn)$ time and determines the minimum total hazard rating you can achieve.

Subproblems

Let $f(i, j)$ be the lowest hazard rating we can achieve on any path from $(1, 1)$ to without having taken the shortcut, and let $g(i, j)$ be the lowest hazard rating haven already taken the shortcut.

Recurrences

We consider any path ending at a cell with a box to have an infinite hazard rating. As the warehouse passed the inspection, there must be some path with a finite total hazard rating, so no such path will be selected.

To reach cell (i, j) with no shortcuts, we can either

- Take the optimal path to cell $(i - 1, j)$ with no shortcuts, then move down, or
- Take the optimal path to cell $(i, j - 1)$ with no shortcuts, then move right.

We should of course choose whichever of these paths has the lesser total hazard rating. This gives the recurrence for f:

$$f(i,j) = \begin{cases} \infty & \text{if } B[i][j], \\ H[i][j] + \min(f(i-1,j), f(i,j-1)) & \text{otherwise.} \end{cases}$$

To reach cell (i, j) with exactly one shortcut, we can

- Take the optimal path to cell $(i - 1, j)$ with exactly one shortcut, then move down,
- Take the optimal path to cell $(i, j - 1)$ with exactly one shortcut, then move right, or
- Take the optimal path to cell $(i - 1, j - 1)$ with no shortcuts, then take the shortcut to (i, j).

Hence, the recurrence for g is:

$$g(i,j) = \begin{cases} \infty & \text{if } B[i][j], \\ H[i][j] + \min(f(i-1,j-1), g(i-1,j), g(i,j-1)) & \text{otherwise.} \end{cases}$$

Base case

$f(1, 1) = H[i][j]$. We can't reach $(1, 1)$ after having taken a shortcut, so $g(1, 1) = \infty$.

Computation

The answer is the minimum hazard rating we can achieve having taken the shortcut, given by $g(m, n)$.

We compute $m \times n$ values for $f(i,j)$ and $g(i,j)$, each in constant time, so the algorithm runs in $O(mn)$. We can compute all values of f before computing all values of g, both in lexicographic order, or we can compute them together in lexicographic order.

An alternative method is to find for each cell (i,j) the minimum total hazard ratings:

- $g(i,j)$, from $(1,1)$ to (i,j), and
- $g'(i,j)$, from (i,j) to (m,n),

solving both subproblems using recurrences similar to those developed in the first two parts. Then the answer is

$$\min_{1 \le i < m, 1 \le j < n} (g(i,j) + g'(i+1, j+1)).$$

Question 3

You have been asked to perform n complex calculations $c_1, \dots, c_n$, where each calculation c_i requires r_i bytes of RAM to store the result. You can perform the calculations *in any order*.

Some calculations depend on the result of others. These dependencies are represented as a directed graph $G = (V, E)$, where E contains an edge $c_j \to c_i$ if the result of c_j has to be in RAM in order to calculate c_i.

Let $\text{pred}(i)$ denote the set of computations that c_i depends on, that is,

$$\text{pred}(i) = \{j : (c_j \to c_i) \in E\}.$$

3.1 [**3 marks**] Explain why it is impossible to perform all n calculations unless the graph G is acyclic.

Let $c_j \Rightarrow c_i$ denote that there is some path from c_j to c_i in G. If $(c_j \to c_i) \in E$, then $c_i \Rightarrow c_j$, as this edge forms a path. Additionally, this relation is transitive: if $c \Rightarrow b$ and $b \Rightarrow a$, then there is a path from c to a created by concatenating the path $c \Rightarrow b$ to the path $b \Rightarrow a$, so $c \Rightarrow a$. If $c_j \Rightarrow c_i$, then c_j must be computed before c_i.

If G is **not** acyclic, then there is some series of calculations $x_1 \to x_2 \to \cdots \to x_k \to x_1$ that forms a cycle in the graph.

Since $x_k \Rightarrow x_1$, calculation x_1 depends directly on x_k, so we must perform x_k before x_1. However, since $x_1 \Rightarrow x_k$, we must perform x_1 before x_k. These two requirements are contradictory, so it is impossible to compute all the calculations in the series.

3.2 [**10 marks**] To perform calculation c_i, we first have to collate the results of $\text{pred}(i)$, which takes

$$\sum_{j \in \text{pred}(i)} r_j$$

seconds. It then takes r_i^2 seconds to compute the result.

On a sequential computer which can only perform one calculation at a time, it takes

$$\sum_{i=1}^{n} \left(r_i^2 + \sum_{j \in \text{pred}(i)} r_j \right)$$

seconds to determine all results. However, we have a massively parallel computer that can perform an unlimited number of calculations at the same time.

Design an algorithm that runs in $O(n+m)$ time and determines the minimum amount of time required to perform all n calculations on the parallel computer, where m is the number of dependencies, i.e. the number of edges in the graph.

For parts 2 and 3, we assume that the sum and max of an empty set are 0.

Preprocessing

Re-index the computations by topological order in the graph, i.e. if $c_j \in \operatorname{pred}(c_i)$, then $j < i$.

Subproblem

Let $\operatorname{opt}(i)$ be the minimum time to perform calculation i (including any others it depends on, directly or indirectly).

Recurrence

To perform calculation i, we first have to wait for all calculations i depends on to complete. The time taken here is that of the slowest calculation in $\operatorname{pred}(i)$. Hence, we have

$$\operatorname{opt}(i) = \max_{j \in \operatorname{pred}(i)} \operatorname{opt}(j)) + r_i^2 + \sum_{j \in \operatorname{pred}(i)} r_j.$$

Note that it is always possible to do calculation i as soon as its predecessors are complete; using the parallel computer, we can do everything required for calculation i on a new thread.

Base cases

Not necessary; if there are no predecessors, then $\operatorname{opt}(i) = r_i^2$, which is the same value obtained by the recurrence.

Order of computation

Each $\operatorname{opt}(i)$ value only depends on the $\operatorname{opt}(\cdot)$ values of its predecessors, so processing in topological order guarantees that $\operatorname{opt}(j)$ is solved for all $j \in \operatorname{pred}(i)$ before we solve $\operatorname{opt}(i)$.

Overall answer

The answer is the minimum amount of time taken to compute the slowest computation, given by $\max\limits_{1 \le i \le n} \operatorname{opt}(i)$.

Time complexity

There are n subproblems, and the time taken to compute $\operatorname{opt}(i)$ is $O(|\operatorname{pred}(i)|)$. Since the sum of $|\operatorname{pred}(i)|$ over all i is m, the algorithm runs in $O(m+n)$.

3.3 [**7 marks**] We now have access to a supercomputer which can compute results instantly. If we choose to use the supercomputer for calculation c_i, it takes only

$$\sum_{j \in \operatorname{pred}(i)} r_j$$

time to collate the previous results, without the additional r_i^2 seconds to compute the result.

The supercomputer is very expensive to run, so it can be used at most s times.

Design an algorithm that runs in $O(s(n+m))$ time and determines the minimum amount of time required to compute all n results using the supercomputer for at most s calculations, and the parallel computer for all other calculations.

The intended answer was as follows.

Subproblem

Let $\operatorname{opt}(i,k)$ be the minimum time to perform calculation i (and all required predecessors), with k uses of the supercomputer remaining.

Recurrence

To perform calculation i, we can either:

- Use the supercomputer for the calculation, and complete all previous calculations with one less supercomputer use left, or
- Perform the calculation on the parallel computer, and complete the previous calculations with the k supercomputer uses still available.

We take whichever of these gives the smallest time. Either way, we also incur the time taken to collate the results of the predecessor calculations.

$$\operatorname{opt}(i,k) = \min\begin{pmatrix} \max\limits_{j \in \operatorname{pred}(i)} \operatorname{opt}(j,k) + (r_i)^2, \\ \max\limits_{j \in \operatorname{pred}(i)} \operatorname{opt}(j,k-1) \end{pmatrix} + \sum_{j \in \operatorname{pred}(i)} r_j$$

Base case

If $k = 0$ we cannot use the supercomputer, so

$$\operatorname{opt}(i,0) = \max_{j \in \operatorname{pred}(i)} (\operatorname{opt}(j,0)) + r_i^2 + \sum_{j \in \operatorname{pred}(i)} r_j.$$

as in part 2.

Order of computation

We compute the subproblem results in increasing order of k, then increasing (topological) order of i. There are s subproblems for each of the n calculations.

Overall answer

The answer is the minimum amount of time taken to compute the slowest computation with all k supercomputer uses available, given by $\max\limits_{1 \le i \le n} \operatorname{opt}(i,k)$.

Time complexity

For any value of k, each subproblem and each dependency is considered exactly once, so the algorithm runs in $O(s(n+m))$ time.

Unfortunately, this answer is actually incorrect! The issue here is in the recurrence. If we have k uses of the supercomputer remaining, we cannot allocate k uses to each of the predecessors. We must instead allocate all k uses between the predecessors, and there could be many (exercise: how many?) ways of doing this. Indeed, to the best of our knowledge, this problem cannot be solved within the target time complexity. We sincerely apologise for this oversight.