Due Friday 10th of November at 6pm Sydney time (week 9)

In this assignment we will apply dynamic programming. There are *three problems* each worth 20 marks, for a total of 60 marks. Partial credit will be awarded for progress towards a solution. We'll award one mark for a response of "one sympathy mark please" for a whole question, but not for parts of a question.

Any requests for clarification of the assignment questions should be submitted using the Ed forum. We will maintain a FAQ thread for this assignment.

For each question requiring you to design an algorithm, you *must* justify the correctness of your algorithm. If a time bound is specified in the question, you also *must* argue that your algorithm meets this time bound. The required time bound always applies to the *worst case* unless otherwise specified.

You must submit your response to each question as a separate PDF document on Moodle. You can submit as many times as you like. Only the last submission will be marked.

Your solutions must be typed, *not* handwritten. We recommend that you use LaTeX, since:

- as a UNSW student, you have a free Professional account on Overleaf, and
- we will release a LaTeX template for each assignment question.

Other typesetting systems that support mathematical notation (such as Microsoft Word) are also acceptable.

Your assignment submissions must be your own work.

- You may make reference to published course material (e.g. lecture slides, tutorial solutions) without providing a formal citation. The same applies to material from COMP2521/9024.
- You may make reference to either of the recommended textbooks with a citation in any format.
- You may reproduce general material from external sources in your own words, along with a citation in any format. 'General' here excludes material directly concerning the assignment question. For example, you can use material which gives more detail on certain properties of a data structure, but you cannot use material which directly answers the particular question asked in the assignment.
- You may discuss the assignment problems privately with other students. If you do so, you must acknowledge the other students by name and zID in a citation.
- However, you must write your submissions entirely by yourself.
  - Do not share your written work with anyone except COMP3121/9101 staff, and do not store it in a publicly accessible repository.
  - The only exception here is UNSW Smarthinking, which is the university's official writing support service.

Please review the UNSW policy on plagiarism. Academic misconduct carries severe penalties.

Please read the Frequently Asked Questions document, which contains extensive information about these assignments, including:

- how to get help with assignment problems, and what level of help course staff can give you
- extensions, Special Consideration and late submissions
- an overview of our marking procedures and marking guidelines
- how to appeal your mark, should you wish to do so.

Question 1 *OurExperience*

You are collecting survey results from k students over n days. Each day, all k students answer the survey by providing a *real* number between 0 and 10 representing their stress level. For each student, their survey responses over the n days are given in an array $S_i[1..n]$.

You want to combine all k survey responses to report to your boss how the students are feeling. You are going to make a report which uses one of the responses from each day, and you want to pick-and-choose responses so that students appear to be emotionally stable. You have designed a heuristic to do so:

> Choose a sequence of survey responses $R[1..n]$, where each $R[i]$ is the response from one of the students on day i, minimising the change between each response in the sequence. That is, choose a sequence R where each $R[i]$ is one of $\{S_1[i], S_2[i], \ldots, S_k[i]\}$ minimising the *fluctuation*
>
> $$f = \sum_{i=2}^{n} |R[i-1] - R[i]| \,.$$

For example, with $n = 4$ and $k = 3$, if you record the responses

$$S_1 = [2, 5, 8, 9.9], \qquad S_2 = [5, 2.5, 1, 4], \qquad S_3 = [10, 3, \pi^2, 7],$$

then the optimal sequence is $R = [5, 5, 8, 7]$ with

$$f = |5 - 5| + |5 - 8| + |8 - 7| = 4,$$

obtained by taking the responses from students $2, 1, 1, 3$.

1.1 **[6 marks]** Consider this attempt to solve the problem with a greedy algorithm:

> Start with $S_1[1]$, then for each subsequent day choose the survey response closest to the previous one. Repeat by starting at each $S_2[1], S_3[1], \ldots, S_k[1]$, and of these, choose the sequence with the minimum fluctuation.

Show that this algorithm does not solve the problem correctly by giving a counterexample. You must provide n, k, the k sequences of survey responses, the answer (R and f) produced by the greedy algorithm, and the correct answer. Your example must have $n \leq 4$ and $k \leq 3$.

Suppose $n = 3, k = 2, S_1 = [5, 1, 1]$ and $S_2 = [6, 8, 11]$.

The greedy algorithm will start with 5 from sensor 1, then choose 8 from sensor 2 (as 8 is closer to 5 than 1 is), then 11 from sensor 2 (11 is closer to 8 than 1 is). This gives the sequence $[5, 8, 11]$, with $f = |5 - 8| + |8 - 11| = 6$.

It will then try again starting with 6 from sensor 2, and will once again choose 8 and 11, giving the sequence $[6, 8, 11]$ with a total change of 5.

Comparing these two options, the greedy algorithm will choose $R = [6, 8, 11]$.

However, we can achieve a better solution by choosing $R = [5, 1, 1]$, taking all measurements from sensor 1. Then $f = |5 - 1| + |1 - 1| = 4$, which is better than the greedy algorithm's solution.

The greedy algorithm does not work since it may be better to take a large difference early in the sequence resulting in less change later on.

1.2 [14 marks] Design an $O(nk^2)$ algorithm to find the minimal fluctuation of an optimal sequence R, and the sequence of survey responses used to achieve this fluctuation. If there are multiple optimal sequences, your algorithm can find any one of them.

Answers that determine only the minimal fluctuation (without finding the sequence of responses) will receive at most 10 marks for this part.

Subproblems. For all $1 \leq i \leq n$ and $1 \leq j \leq k$, let $\text{opt}(i, j)$ be the minimum f for a sequence of the first i measurements ending in a measurement from sensor j, and $\text{prev}(i, j)$ the previous sensor used in the sequence.

Recurrence. To form a sequence ending with $S_j[i]$, we can extend any sequence ending with one of the $S_a[i-1]$ values for any a. Since any choice results in a valid sequence, we should choose the one minimising the total cost - that is, the cost of the sequence up to $i-1$, plus the difference between the $i-1$th measurement and $S_j[i]$.

That is,

$$\text{opt}(i, j) = \min_{1 \leq a \leq k} \text{opt}(i-1, a) + |S_a[i-1] - S_j[i]|$$

$$\text{prev}(i, j) = \arg\min_{1 \leq a \leq k} \text{opt}(i-1, a) + |S_a[i-1] - S_j[i]|$$

Base case. For $i = 1$ each sequence only has one possible value, so the total difference is 0. That is, for all j, $\text{opt}(1, j) = 0$. There is no previous measurement, so $\text{prev}(1, j)$ is undefined.

Final answer. The minimum f is the minimum of any sequence of the first n days ending with any sensor, that is $\min_{1 \leq a \leq k} \text{opt}(n, a)$. The optimal sequence therefore ends with sensor $S = \arg\min_{1 \leq a \leq k} \text{opt}(n, a)$, and is formed recursively as follows: if S' is used for the jth measurement, then $\text{prev}(j, S')$ is used for the $(j-1)$th measurement. Applying this rule $n-1$ times gives the full sequence of sensors used in the optimal sequence.

Order of computation. Each subproblem depends only on the previous value of i, so we solve in increasing order of i then j.

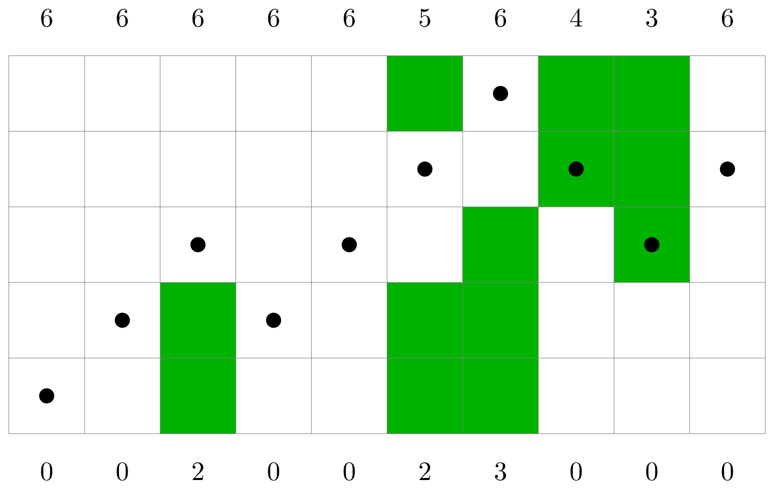
Time complexity. There are nk subproblems, each of which is solved in $O(k)$ time by taking the minimum over k previous subproblems. Hence, the recurrence takes $O(nk^2)$ time. The final answer is found by backtracking through the prev sequence in $O(n + k)$ time, so the algorithm runs in $O(nk^2)$.

Question 2 *Crappy Bird*

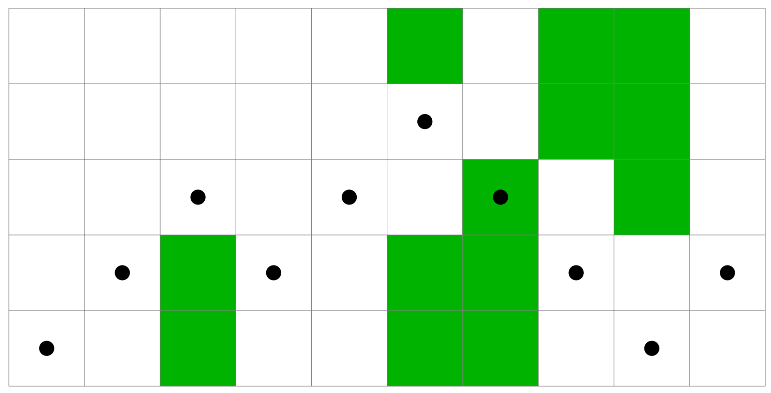
Crappy Bird is a game where you control a bird starting at $(1, 1)$ in a grid with m rows of n columns (where $m \geq 2$). Some columns j $(1 \leq j \leq n)$ have pipes from the bottom of the screen to row $\mathsf{PipeUp}(j)$, and some have pipes from the top of the screen to row $\mathsf{PipeDown}(j)$. If a column j has no pipe from the bottom then $\mathsf{PipeUp}(j) = 0$, and if it has no pipe from the top then $\mathsf{PipeDown}(j) = m + 1$. The goal is to reach column n while hitting as few pipes as possible. If you exit the screen by flying too low or too high, you lose the game immediately!

Each step of the game, the bird moves one column to the right. Additionally, if you tap the screen the bird moves one row up, otherwise it falls one row down. That is, if the bird is at (x, y) at the some step, then it can be at $(x + 1, y + 1)$ or $(x + 1, y - 1)$ in the next step.

For example, this is a valid path which hits two pipes. The PipeUp and PipeDown values are shown below and above the grid respectively.



A better path is achieved by going down from $(6, 4)$ instead of up, hitting only one pipe.



2.1 **[10 marks]** Design an algorithm that runs in $O(mn)$ time, and determines the fewest pipes the bird can run into without exiting the screen.

Let $\mathsf{Pipe}(i, j)$ be 1 if the cell (i, j) coincides with a pipe, and 0 otherwise. That is,

$$\mathsf{Pipe}(i, j) = \begin{cases} 0 & \text{if } \mathsf{PipeUp}(i) < j < \mathsf{PipeDown}(i), \\ 1 & \text{otherwise.} \end{cases}$$

Subproblems. For all $1 \leq i \leq n$ and $1 \leq j \leq m$, $\text{opt}(i,j)$ is the smallest number of pipes the bird can hit to get to cell (i,j).

Recurrence. To get to cell (i,j), the bird must have come from cell $(i-1,j-1)$ (if the screen was tapped) or cell $(i-1,j+1)$ (otherwise). Out of these two options, we should obviously choose the one we can get to by hitting the fewest pipes. If cell (i,j) has a pipe, then we also include it in the number of pipes hit.

This gives the recurrence: for all $2 \leq i \leq n$ and $1 \leq j \leq m$,

$$\text{opt}(i,j) = \min(\text{opt}(i-1,j-1), \text{opt}(i-1,j+1)) + \mathsf{Pipe}(i,j)$$

Base cases.

We start at cell $(1,1)$, so we can't hit any previous pipes to get there, giving $\text{opt}(1,1) = \mathsf{Pipe}(1,1)$.

Any other cell in column 1 is unreachable, so $\text{opt}(1,j \neq 1) = \infty$. Similarly, we lose the game if the bird leaves the screen, so $\text{opt}(i,0) = \text{opt}(i,m+1) = \infty$.

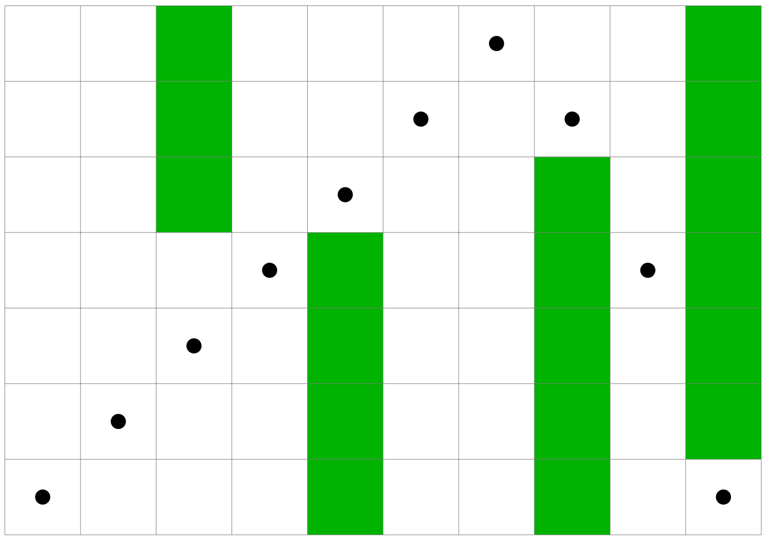
Final answer. The answer is the fewest pipes we can hit to get to any cell in column n, given by $\min_{1 \leq j \leq m} \text{opt}(n,j)$.

Order of computation. Each subproblem depends only on the previous column $i-1$, so we solve in increasing order of i then j.

Time complexity. There are mn subproblems, each solved in constant time, and the final answer takes $O(m)$ time to compute. Hence, the algorithm runs in $O(mn)$ time.

2.2 [**7 marks**] The developers have realised that the game's implementation of gravity is entirely unrealistic! To fix it, they have patched the game to include acceleration for downwards movement. When the bird begins falling (after a sequence of upwards movements), it first falls 1 unit downwards. If it falls again in the next step, it falls 2 units, then 3, 4, and so on. That is, the kth consecutive time the bird falls, it moves from (x,y) to $(x+1,y-k)$.

For example, the bird may take this path:



Note that no pipes are hit, as at each step the bird is in a cell that does not coincide with a pipe. **The movement between positions is not considered.**

Design an algorithm that runs in $O(m^2n)$ time, and determines the fewest pipes the bird can run into.

If we have just fallen 0 times we came from the row below, and could have gotten to the previous cell after any number of falls:

$$\mathrm{opt}(i, j, 0) = \min_{0 \le x \le m} \mathrm{opt}(i-1, j-1, x) + \mathsf{Pipe}(i, j)$$

If we have fallen $k > 0$ times previously, then we must have come from row $j + k$, and gotten there by falling $k - 1$ times previously.

$$\mathrm{opt}(i, j, k) = \mathrm{opt}(i-1, j+k, k-1) + \mathsf{Pipe}(i, j)$$

There are mn subproblems where $k = 0$, each taking $O(m)$; and m^2n subproblems where $k > 0$, each taking $O(1)$. Hence, the time is $mn \cdot O(m) + m^2n \cdot O(1) = O(m^2n)$.

2.3 **[3 marks]** Describe how the algorithm for part 2.2 can be improved to $O(nm\sqrt{m})$. You do not need to restate the entire algorithm - just describe and justify how you would change the algorithm to achieve this.

You may choose the skip this part and instead give an $O(nm\sqrt{m})$ algorithm for part 2. If you do, your answer to 2.2 will be marked for both parts.

If the bird falls for k consecutive turns, it will fall $1 + 2 + 3 + \cdots + k$ units downwards, and $1 + 2 + 3 + \cdots + k < m$ as we can't fall off the bottom edge. So,

$$\begin{aligned}
1 + 2 + 3 + \cdots + k &< m \\
k(k+1)/2 &< m && \text{(arithmetic series)} \\
k^2 + k &< 2m \\
k^2 &< 2m \\
k &< \sqrt{2m} \\
k &< \sqrt{2}\sqrt{m} \\
k &< 2\lceil\sqrt{m}\rceil.
\end{aligned}$$

So if we solve $\mathrm{opt}(i, j, k)$ for $0 \le k \le 2\lceil\sqrt{m}\rceil$, we will include every number of falls that can actually occur. Therefore, we only need to solve $2nm\lceil\sqrt{m}\rceil = O(nm\sqrt{m})$ subproblems, and for subproblems where $k = 0$, we only need to take the minimum over $0 \le x \le 2\lceil\sqrt{m}\rceil$. This change gives a time complexity of $O(nm\sqrt{m})$.

Question 3 *Dance!*

In the arcade game *Dance Dance Revolution* (DDR), players stand on a stage and hit arrows as they scroll across the screen. More specifically, a sequence of n arrows (⬅, ⬆, ➡, ⬇) will scroll across the screen, and as each arrow hits the top of the screen, the player must stand on the corresponding arrow on the stage.

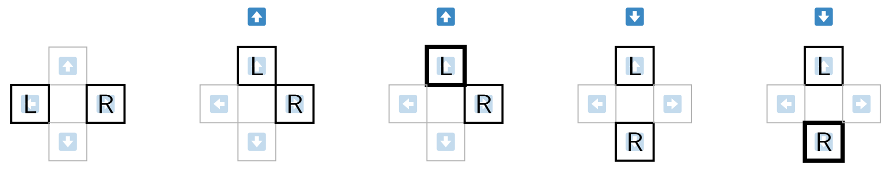
We play a variant of DDR, aptly named *Don't Dance Revolution* (DDR2), where the goal is to play the game like DDR but move as little as possible. The game plays like DDR except when an arrow reaches the top of the screen and the player already has a foot on the correct arrow, then the player is awarded one lazy point. If neither foot is on the correct arrow, then the player must move *exactly* one foot from its current location to the correct arrow on the platform.

Unfortunately, the game is a bit unforgiving: any wrong move will cause the player to lose the game and *all* of their lazy points. Wrong moves include:

- Failing to step on the correct arrow.
- Moving more than one foot at any given time.
- Moving either foot when the player is already stepping on the correct arrow.

You are given a sequence A of n arrows. Assume that your left foot starts on ⬅ and your right foot starts on ➡, and you have memorised the entire sequence of arrows.

For example, consider the following sequence: ⬆ ⬆ ⬇ ⬇. We can earn up to two lazy points as follows:



Start game! *Lazy point!* *Lazy point!*

3.1 [**4 marks**] Show that it is always possible to earn at least $\lfloor n/4 \rfloor$ lazy points during a round of DDR2.

Split the sequence of n arrows into $\lfloor n/4 \rfloor$ blocks of size 4, followed by the remaining $n \pmod 4$ arrows. We now argue that it is possible to earn a lazy point in each block of four.

- **Case 1**: There exist some arrow that appears more than once in a block of four. When the arrow appears for the first time, step on the arrow with your left foot, and then only move the right foot for the rest of the block. When the arrow appears again, you earn one lazy point.
- **Case 2**: Each arrow appears exactly once in the block of four. Only move your right foot for the duration of the block. When the arrow appears under your left foot, you earn one lazy point.

In both of these cases, you earn at least one lazy point in each block and hence, it is always possible to earn at least $\lfloor n/4 \rfloor$ lazy points during a round of DDR2 even when you start with the left and right arrows.

3.2 [**16 marks**] Design an $O(n)$ algorithm to determine the maximum number of lazy points you can earn in a round of DDR2.

We solve this with dynamic programming. Let $\mathsf{point}(i, L, R)$ denote the maximum possible score starting *after* at the ith arrow with the left foot on L and the right foot on R. To derive a suitable recursion, we need to make a few observations:

- **Observation 1**: If $i = n$, then there is no score possible. This defines a suitable base case for our recursion.
- **Observation 2**: If our left or right feet are already on the correct arrow, then the maximum possible score starting just after the ith arrow is one more than the maximum possible score starting just after the $(i+1)$-th arrow; therefore, in this case, we obtain the recursion
$$\mathsf{point}(i, L, R) = 1 + \mathsf{point}(i+1, L, R),$$
as long as $A[i+1] \in \{L, R\}$.
- **Observation 3**: Otherwise, we consider separately if we step with our left foot or our right foot, giving us
$$\mathsf{point}(i, L, R) = \max\{\mathsf{point}(i+1, A[i+1], R), \mathsf{point}(i+1, L, A[i+1])\}.$$

For $0 \le i < n$, this gives the following recursion:

$$\mathsf{point}(i, L, R) = \begin{cases} 1 + \mathsf{point}(i+1, L, R) & \text{if } A[i+1] \in \{L, R\}, \\ \max\left\{ \begin{array}{c} \mathsf{point}(i+1, A[i+1], R), \\ \mathsf{point}(i+1, L, A[i+1]) \end{array} \right\} & \text{otherwise.} \end{cases}$$

The base case is $\mathsf{point}(n, L, R) = 0$ for each $L \in \{\uparrow, \rightarrow, \downarrow, \leftarrow\}$ and $R \in \{\uparrow, \rightarrow, \downarrow, \leftarrow\}$.

The final solution is $\mathsf{point}(0, \leftarrow, \rightarrow)$, the maximum lazy points starting after the 0th arrow with the left foot on $\leftarrow$ and right foot on $\rightarrow$.

Since L and R are bounded to four options, we can memoise this in an $(n+1) \times 4 \times 4$ table which we can fill bottom-up, giving us an $O(n)$ solution.

Each subproblem $\mathsf{point}(i, L, R)$ depends only on subproblems $\mathsf{point}(i+1, \cdot, \cdot)$, so we solve in decreasing order of i and any order of L and R.

The problem can also be solved by considering the maximum score *up to* the ith arrow by modifying the recurrence appropriately.