Due Friday 28th of July 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

You are travelling with your friends along the Geraldton coast with cities $c_0, c_1, c_2, \ldots, c_n$ on the shore. You are starting in city c_0 where a famous spa is, and need to reach the airport situated in city c_n, so you will visit each city $c_0, c_1, c_2, \ldots, c_n$ in that order.

In each city you may swap the animal you are riding on and the choices are a giraffe, a mammoth, an ant, an iguana, and a lemur, denoted G, M, A, I, L respectively. However, each city has its own rules what kind of animal exchanges are allowed. For example, in some of the cities you can swap a giraffe only for a lemur or an ant (and you **cannot** remain on your giraffe), in others you can swap a mammoth only for a giraffe or decide to remain on your mammoth, and so on. You know all the rules of all the cities $c_1, \ldots, c_n$, expressed by a function $R(i, a, b)$ where $R(i, a, b) = 1$ if in city c_i one can swap animal a for animal b, and zero otherwise (a and b belong to the set $\{G, M, A, I, L\}$, and $1 \leq i < n$). You also know the speed $v(a)$ in km/h ($a \in \{G, M, A, I, L\}$) of each of the five animals, as well as the distances $d(i)$ in km between cities c_{i-1} and c_i for all $i = 1, 2, \ldots, n$. Calculating a given value of R, v, or d can be done in $O(1)$ time.

You may begin your journey from c_0 on any of the five animals. You need to choose which animals to use for each of the n trips between cities, such that your travel time is minimised and every animal swap is valid.

1.1 [**2 marks**] Consider the case of $n = 2$, with $d(1) = 1$, $d(2) = 3$, and v defined as

$$v(G) = 7, \qquad v(M) = 3, \qquad v(A) = 2, \qquad v(I) = 1, \qquad v(L) = 4.$$

Define R so that

$$R(1, G, M) = 1, \quad R(1, A, M) = 1, \quad R(1, A, A) = 1, \quad R(1, I, G) = 1, \quad R(1, L, I) = 1,$$

and $R(1, a, b) = 0$ for all other values of a, b.

Determine which animals you should choose on each trip in order to minimise the total time taken to travel from c_0 to c_2. You *must* justify your selection.

Considering the compatibilities given by R, the only valid trips are:

Giraffe then Mammoth	$1/7 + 3/3 \approx 1.14$
Ant then Mammoth	$1/2 + 3/3 = 1.5$
Ant then Ant	$1/2 + 3/2 = 2$
Iguana then Giraffe	$1/1 + 3/7 \approx 1.43$
Lemur then Iguana	$1/4 + 3/1 = 3.25$

Of these, Giraffe from c_0 to c_1 and Mammoth from c_1 to c_2 is the fastest.

1.2 [**12 marks**] Design an algorithm which determines the minimal amount of time (in hours) it will take to get from c_0 to c_n without making invalid swaps, as well as the animals required on each trip to achieve this time. Your algorithm must run in $O(n)$ time.

[A] For each i and each animal $a = G, M, A, I, L$ we solve the problem "$P(i, a) =$ *the minimal amount of travel time needed to arrive to city c_i riding the last leg of your journey (from c_{i-1} to c_i) on animal a*".

[B] Let $\text{opt}(i, a)$ denote the minimal amount of travel time needed to reach city c_i riding the last leg on animal a, and let $A(i, a)$ be the animal which you swapped for animal a in city c_{i-1} during such fastest journey.

The base cases are given by $\text{opt}(1, a) = d(1)/v(a)$ for $a = G, M, A, I, L$. For $2 \leq i \leq n$ we have recursion

$$\mathrm{opt}(i,a) = \min\{\mathrm{opt}(i-1,x) + {}^{d(i)}/_{v(a)} \;:\; R(i-1,x,a)=1; x = G,M,A,I,L\};$$
$$A(i,a) = \arg\min\{\mathrm{opt}(i-1,x) + {}^{d(i)}/_{v(a)} \;:\; R(i-1,x,a)=1; x = G,M,A,I,L\}.$$

[C] The final solution is given by $\min\{\mathrm{opt}(n,x) : x = G,M,A,I,L\}$; to obtain an optimal swapping strategy we let $\alpha(n) = \arg\min\{\mathrm{opt}(n,x) : x = G,M,A,I,L\}$, and then backtrack, i.e., define recursively $\alpha(n-i) = A(n-i, \alpha(n-(i-1)))$ for all $1 \le i \le n$. Note that $\alpha(0)$ is the animal you should start your journey with in c_0.

[D] Clearly, the procedure takes $O(n)$ many steps.

1.3 [6 marks] While planning your trip, your friend Mae points out that animals are living creatures, and do, in fact, need to rest. To take this into consideration, you estimate the effect of consecutive trips on the animals, and come up with a magical constant, $0 < \varepsilon < 1$. For each consecutive trip an animal makes, the speed of the animal is multiplied by this constant.

For example, if you decide to travel on a mammoth from c_0 to c_1 to c_2 to c_3 without ever changing animals, then the mammoth's speed will be $v(M)$ from c_0 to c_1, $\varepsilon v(M)$ from c_1 to c_2, and $\varepsilon^2 v(M)$ from c_2 to c_3. Of course, if you then swap animals, and later decide to travel by mammoth again, the new mammoths are not tired, and thus have speed $v(M)$. You may not "swap" an animal for "new" animals of the same type.

Design an algorithm which determines the minimal amount of time (in hours) it will take to get from c_0 to c_n without making invalid swaps. Your friends are judging you for not considering this in the first place, so your algorithm must run in $O(n^2)$ time before things become more awkward.

The fastest way to reach city i may involve taking a slower than optimal sequence of animals to city $i-1$. However, if we consider only the routes that end in a sequence of the same animal a certain number of times, we can optimally solve a more restricted subproblem.

[A] For each i, k and each animal $a = G,M,A,I,L$ we solve the problem "*$P(i,k,a)$ = the minimal amount of travel time needed to arrive to city c_i riding the last leg of your journey (from c_{i-1} to c_i) on animal a, and having ridden the same animal for k previous legs (inclusive of this one)*".

[B] Let $\mathrm{opt}(i,k,a)$ denote the minimal amount of travel time needed to reach city c_i riding the last leg on animal a, with the same animal ridden k times.

The base cases are given by $\mathrm{opt}(1,1,a) = {}^{d(1)}/_{v(a)}$ for all a, and $\mathrm{opt}(1,k,a) = \infty$ for all a and $1 < k \le n$. For $2 \le i \le n$ and $1 \le k \le i$ we have two cases for recursion. If $k \ge 2$,

$$\mathrm{opt}(i,k,a) = \begin{cases} \infty, & \text{if } R(i-1,a,a) = 0; \\ \mathrm{opt}(i-1,k-1,a) + {}^{d(i)}/_{\varepsilon^{k-1}v(a)}, & \text{otherwise.} \end{cases}$$

Otherwise, if $k = 1$,

$$\mathrm{opt}(i,1,a) = \min_{x,k'}\{\mathrm{opt}(i-1,k',x) + {}^{d(i)}/_{v(a)} : R(i-1,x,a) = 1, x \ne a, 1 \le k' < i\}.$$

[C] The final solution is given by $\min_{k,a}\{\mathrm{opt}(n,k,a)\}$.

[D] For $k > 1$ the subproblems take $O(1)$ time to solve, and there are $O(n^2)$ such subproblems. For $k = 1$ the subproblems take $O(n)$ time to solve, and there are $O(n)$ such subproblems. So, overall, the time complexity is $O(n^2)$

Question 2

You are given a string $w = w_1 w_2 \ldots w_n$ of n letters that come from a fixed alphabet. A *palindrome* is a substring w' such that w' can be read the same forwards and backwards. For example, $w' = kayak$ forms a palindrome while $w'' = kayaak$ does not form a palindrome. A *palindromic substring* is a contiguous subsequence of w that forms a palindrome.

2.1 [**8 marks**] We want to eventually find a minimum cost decomposition of w into palindromes. However, to do this, we need to first find *where* the palindromes are. Given a string $w = w_1 \ldots w_n$, describe an $O(n^2)$ algorithm to identify all of the palindromic substrings of w.

Hint. How many subproblems can you have at most?

- There are faster algorithms such as a modification to *Manacher's algorithm* but if you use the algorithm, you will have to state the entire algorithm, and prove its correctness and running time.

Let $\mathsf{palindrome}(i, j)$ denote whether the substring $w_i \ldots w_j$ forms a palindrome. We note that $w_i \ldots w_j$ is a palindrome if and only if $w_i = w_j$ *and* $w_{i+1} \ldots w_{j-1}$ forms a palindrome. Therefore, if $w_i \neq w_j$, then we can return false. Otherwise, we return the result of $\mathsf{palindrome}(i+1, j-1)$. This can be compactly expressed as

$$\mathsf{palindrome}(i, j) = (w_i = w_j) \wedge \mathsf{palindrome}(i+1, j-1).$$

The base cases occur with $\mathsf{palindrome}(i, i) = \mathsf{true}$ for each $i = 1, \ldots, n$ and $\mathsf{palindrome}(i, j) = \mathsf{false}$ if $j < i$. We order the subproblems in increasing order of $j - i$; that is, we solve the subproblems in increasing order of size of the substrings.

We, thus, have an $O(n^2)$-sized dynamic programming table where $\mathsf{palindrome}(i, j)$ determines whether the substring $w_i \ldots w_j$ forms a palindrome. We return all of the subproblems that return true.

To compute each subproblem, we just require an $O(1)$ call to the dynamic programming table. Therefore, the dynamic programming solution takes $O(n^2)$ to construct the table. We then read all of the $O(n^2)$ cells, which takes $O(n^2)$ in total.

2.2 [**12 marks**] A *palindromic decomposition* of a string w is a partition of w into substrings $u_1, \ldots, u_m$ such that $w = u_1 \ldots u_m$ and each part u_i forms a palindrome. For example, if $w =$ `abcbabaab`, then $u_1 =$ `a`$, u_2 =$ `bcb`$, u_3 =$ `a`$, u_4 =$ `baab` is a palindromic decomposition of w.

We assign each substring u_i a cost denoted by $\mathsf{cost}(|u_i|)$, where $|u_i|$ is the length of u_i. Finally, we define the cost of a decomposition $u_1, \ldots, u_m$ to be the sum of the costs of the individual substrings u_i; that is, if $w = u_1 \ldots u_m$, then the cost of the decomposition is

$$\mathsf{cost}(|u_1|) + \cdots + \mathsf{cost}(|u_m|).$$

We want to find the minimum cost of a *palindromic decomposition* of w. Given a string $w = w_1 \ldots w_n$ and an array of all costs, design an $O(n^2)$ algorithm to compute the minimum cost of a palindromic decomposition of w.

Hint. Perform a pre-processing step using **2.1**.

We begin by performing some preprocessing as before by finding all palindromes of w in $O(n^2)$ time. Let $\mathsf{palindrome}(i,j)$ denote whether the substring $w_i \dots w_j$ is a palindrome. Again, we define $C(i,j)$ as follows:

$$C(i,j) = \begin{cases} \mathsf{cost}\left(|w_i \dots w_j|\right) & \text{if } \mathsf{palindrome}(i,j) = \mathsf{true}, \\ \infty & \text{otherwise.} \end{cases}$$

Let $\text{opt}(i)$ be the minimum cost of a palindrome decomposition of the substring $w_1 \dots w_i$. The recurrence is similar to Tutorial 6, Problem 3. Observe that, to split the string $w_1 \dots w_i$ into palindromes, we require that some substring $w_{j+1} \dots w_i$, for some $j < i$, be a palindrome and the remaining string $w_1 \dots w_j$ is the minimum cost decomposition to split $w_1 \dots w_j$ into palindromes. This gives us

$$\text{opt}(i) = \min_{0 \le j < i} \left\{ \text{opt}(j) + C(j+1, i) \right\}.$$

This gives us the following decomposition.

$$\underbrace{w_1 \dots w_j}_{\text{opt}(j)} \quad \underbrace{w_{j+1} \dots w_i}_{\text{palindrome of size } j-i}$$

The base case is $\text{opt}(0) = 0$ with the final solution being $\text{opt}(n)$. We solve the subproblems in increasing order of i. To analyse the time complexity, we observe that there are n subproblems. Each subproblem has at most $O(n)$ work since we need to check at most n subproblems. Therefore, the running time is $O(n^2)$.

Question 3

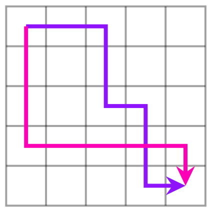
You are participating in a robotics competition, and need to program a robot to move through an m by n grid. The robot starts at cell $(1, 1)$, and must reach cell (m, n) to finish.

Your robot accepts two types of instructions:

- Start moving down at cell (i, j),
- Start moving right at cell (i, j).

An instruction is *not* required to tell the robot which way to move at the start, as there is only one possibility based on the first turning instruction.

For example, in this grid with $m = n = 5$, the purple path requires four instructions (down at $(1, 3)$, right at $(3, 3)$, down at $(3, 4)$, right at $(4, 4)$) while the pink path requires two (right at $(4, 1)$, down at $(4, 5)$).



3.1 [**6 marks**] Design an algorithm that runs in $O(mn)$ time and determines the number of distinct paths through the grid that the robot can traverse.

Hint. A closed form solution is possible, but we naturally encourage the use of Dynamic Programming which will assist with the rest of the question. A similar problem will also be discussed in the Week 8 Tutorial. Consider the 4 ways to reach a cell (i, j).

DP Solution

Subproblems

Let $P(i, j)$ be the number of distinct paths from $(1, 1)$ to (i, j).

Recurrence

To reach cell (i, j) we can either

- Go to cell $(i - 1, j)$, then move down to cell (i, j), or
- Go to cell $(i, j - 1)$, then move right to cell (i, j).

Clearly, these cases are disjoint and cover all possible paths.

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

$$P(i, j) = P(i - 1, j) + P(i, j - 1).$$

Base cases

There is exactly one path to $(1, 1)$, so $P(1, 1) = 1$.

There is only one way to reach a cell in the top row (continuously moving right), so $P(1, j) = 1$. Similarly $P(i, 1) = 1$.

Final answer

The answer is the number of paths to (m, n), given by $P(m, n)$.

Order of computation

We solve in increasing order of i then j, i.e. lexicographical order.

Time complexity

We solve mn subproblems, each in $O(1)$, so the algorithm runs in $O(mn)$.

Closed Form Solution

Any path through the grid must contain $m-1$ moves down and $n-1$ moves right. Each unique arrangement of these moves gives a unique path through the grid. The number of paths is then given by the permutations with repetitions formula:

$$\frac{(m+n-2)!}{(m-1)!(n-1)!}$$

Evaluating the factorials requires $O(m+n) \subset O(mn)$ multiplications.

3.2 [**8 marks**] Unfortunately due to rising inflation your robot now only has a limited amount of memory, and can only store r instructions. Design an algorithm that runs in $O(mnr)$ time and determines the number of distinct paths through the grid that the robot can traverse using at most r instructions.

There are four ways to reach a cell (i, j) when you consider the second last and third last cells in the path. Two of these ways use an instruction, while the other two don't.

Subproblems

Let $P(i, j, k, d)$ be the number of paths from $(1, 1)$ to (i, j) which uses at most k instructions, and reach cell (i, j) by moving in direction $d \in \{\rightarrow, \downarrow\}$.

We can equivalently define two separate subproblems $P(i, j, k)$ and $Q(i, j, k)$, where $P(i, j, k)$ is the number of paths ending in a right move and $Q(i, j, k)$ is the number ending in a down move.

Recurrences

There are four disjoint ways we can reach (i, j):

- First reach $(i-1, j)$ by moving down, then continue moving down to reach (i, j)
- First reach $(i, j-1)$ by moving right, then continue moving right to reach (i, j)
- First reach $(i-1, j)$ by moving right, then switch to moving down to reach (i, j)
- First reach $(i, j-1)$ by moving down, then switch to moving right to reach (i, j).

The first two options do not require an instruction as they continue in the previous direction, while the last two require instructions. To ensure that we are able to make the turn to reach (i, j), we must ensure that at most $k-1$ instructions are used to reach the previous cell.

This gives our recurrences: For all $1 < i \leq m$, $1 < j \leq n$ and $1 < k \leq r$,

$$P(i, j, k, \rightarrow) = P(i, j-1, k, \rightarrow) + P(i, j-1, k-1, \downarrow)$$
$$P(i, j, k, \downarrow) = P(i-1, j, k, \downarrow) + P(i-1, j, k-1, \rightarrow)$$

Base cases

There is only one path from $(1, 1)$ to $(1, 1)$ - the empty path - so $P(1, 1, k, d) = 1$.

The only way to reach a cell in row 1 is by continuously moving right, so $P(1, j, k, \rightarrow) = 1$ and $P(1, j, k, \downarrow) = 0$. Similarly, $P(i, 1, k, \downarrow) = 1$ and $P(i, 1, k, \rightarrow) = 0$.

Finally, it is impossible to reach any cell that is not in row or column 1 without any instructions, so $P(i, j, 0, d) = 0$.

Final answer

The final answer is the number of ways to reach (m, n) using at most r instructions, reaching (m, n) from either direction. Therefore, the answer is given by $P(m, n, r, \rightarrow) + P(m, n, r, \leftarrow)$.

Order of computation

Solve in increasing order of k, i then j, solving for both directions for each (i, j, k). That is,

for $k = 1 \ldots r$ { for $i = 1 \ldots m$ { for $i = 1 \ldots n$ { solve $P(i, j, k, \rightarrow)$, $P(i, j, k, \downarrow)$ } } }.

Time complexity

We solve $2mnr$ subproblems, each in $O(1)$ time, so the algorithm runs in $O(mnr)$.

3.3 **[6 marks]** To make the competition more interesting, Blake has placed obstacles in some of the cells, which the robot is unable to pass through. They have given you an array $O[1 \ldots m][1 \ldots n]$, where $O[i][j]$ is TRUE if cell (i, j) has an obstacle and FALSE otherwise. Of course, Blake has made sure that a robot can travel from $(1, 1)$ to (m, n) by moving only right and down.

You wish to find the smallest amount of memory r that your robot needs to be able to traverse the grid.

Design an algorithm that runs in $O(mnr)$ time and finds the smallest number of instructions needed to traverse the grid.

Note that the parameter r in the time complexity is also the answer your algorithm is trying to find. This is similar to the Ford-Fulkerson algorithm for max flow, where the time complexity is dependent on the max flow for the given graph.

An algorithm that runs in $O(mn(m + n))$ will be eligible for at most 4 marks for this part.

We adjust the answer to part 1, by adding the condition that $P(i, j, k, d) = 0$ if $O[i][j]$ is TRUE, as it is impossible to end on a cell containing an obstacle.

Recurrences

For all $1 < i \leq m$, $1 < j \leq n$ and $1 < k \leq r$,

$$P(i,j,k,\rightarrow) = \begin{cases} 0, & \text{if } O[i][j]; \\ P(i,j-1,k,\rightarrow) + P(i,j-1,k-1,\downarrow), & \text{otherwise.} \end{cases}$$

$$P(i,j,k,\downarrow) = \begin{cases} 0, & \text{if } O[i][j]; \\ P(i-1,j,k,\downarrow) + P(i-1,j,k-1,\rightarrow), & \text{otherwise.} \end{cases}$$

The base cases are unchanged.

Computation and answer

We solve in increasing order of k, i then j as in part 1. However, we stop immediately at the first value of k for which $P(m,n,k,\rightarrow) + P(m,n,k,\downarrow) \geq 1$, because this is the first time we can reach the end, and hence must be the least number instructions possible to reach the end, so the value of k when this occurs must be r, in which is our answer.

Time complexity

For each value of k, we solve $2mn$ subproblems, each in constant time. We only run our algorithm until reaching the end is possible, which occurs when k is r, so there must be $2mnr$ subproblems, each of which taking $O(1)$ time. So the algorithm runs in $O(mnr)$.