General judging notes

- Basic format: students (in groups of 1, 2 or 3) will come from their table to you with their solution to a problem, and explain it to you. They can also come if they're stuck and need some help. If you think they should move on to the next problem, hand it to them (printed on a piece of paper), along with a Mentos, and they'll go back to their table.
- Be helpful and encouraging. Some weaker students will likely attend, so identify their strengths and make them feel good about them.
- If a student/group presents a solution that doesn't quite work, preferably they will fix it on the spot, or go back to their table to think more, and you can provide a hint if they want one. Alternatively, if you think it's more appropriate (e.g. they seem stressed), consider letting them move on to the next problem.
- Ask for clarification if their algorithm is unclear, it should be a two-way dialogue.
- Ask them to state and justify their algorithm's time complexity.
- Ask for an informal justification of their algorithm's correctness (since proving correctness is a key skill learned in the course).
- Adjust the level of detail and rigour required to move on based on the length of the queue. At the beginning the queue will likely build up, so let people move on very quickly.
- Feel free to ask other tutors (Ryan and Isaiah are familiar with all the problems) if you need any assistance when judging.
- If the queue is empty, feel free to get up and talk to students at their tables and see how they're going. Or join another tutor and judge with them.

Question 1 *Blocks*

You are given n stacks of identical blocks. The ith stack contains a positive number of blocks, let us denote this as h_i. You are also able to move any number of blocks from the ith stack to the $(i+1)$th stack, as long as every stack always contains a positive number of blocks. You want to know if the sizes of the stacks can be made *strictly* increasing. For example $\langle 1, 3, 6, 8\rangle$ is *strictly* increasing, but $\langle 1, 4, 4, 7\rangle$ is not.

Design an $O(n)$ algorithm that determines whether it is possible for the stacks to be made *strictly* increasing.

Use the greedy method. What is the fewest number of blocks required in the ith stack of a *strictly* increasing sequence?

In a sequence of stacks with strictly increasing height, the kth stack have have height at least k blocks, and therefore the first k stacks must have at least $k(k+1)/2$ blocks.

Claim: a sequence can be made strictly increasing if and only if

$$\sum_{i=1}^{k} h_i \geq \frac{k(k+1)}{2}$$

for all $1 \leq k \leq n$.

Proof:

If this inequality fails for some k, then the first k stacks cannot be made strictly increasing in height. Since blocks can be moved to later stacks but not earlier stacks, this ensures that the n stacks cannot be made strictly increasing.

However, if this inequality is true for all k, then we can make the blocks strictly increasing using the following algorithm. Traverse the stacks from first to last, at each stage keeping exactly k blocks in the kth stack and moving the rest to stack $k+1$. The assumed inequality ensures that when we get to stack k, it will always have at least

$$\frac{k(k+1)}{2} - \frac{(k-1)k}{2} = k$$

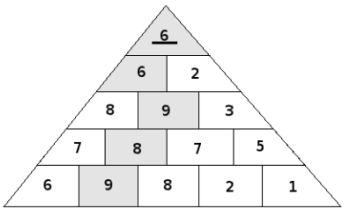
blocks in it. At the last stack, no moves are available, so leave the stack as is.

Thus, we can make the sequence strictly increasing if and only if the sum of the first k initial heights is at least $k(k+1)/2$ for all $1 \leq k \leq n$. We can check all of these inequalities in one pass of the sequence, by maintaining the sum of the first k values h_i. Each new stack adds one term to the sum, taking $O(1)$ to update the sum and $O(1)$ to check it. The total complexity is $O(n)$.

- You can solve this problem without dealing with the $k(k+1)/2$ stuff, just sweep left to right and greedily move as many blocks as you can.
- Ask for a justification as to why their greedy works just to keep them on their feet, but don't make them give more than a two sentence answer, so the queue keeps moving.

Question 2 *Triangle*

You are given a triangular grid of positive integers. The grid consists of n rows, the ith of which has i entries. For $1 \leq j \leq i \leq n$, let $T(i, j)$ denote the jth entry in row i.



Define a *route* to be any path that starts at the top entry and ends at any entry of the bottom row, with each step going either diagonally down to the left or diagonally down to the right. Your task is to find the largest sum of numbers that can be encountered on a route. For example, in the pictured triangular grid, the optimal route is indicated by grey cells, and so the answer is $6 + 6 + 9 + 8 + 9 = 38$.

Design a dynamic programming algorithm which solves the task in $O(n^2)$ time.

Observe that a path from the top down to $T(i, j)$ must take some entry from row $i-1$. There are at most two options for which entry to take.

Subproblems: for $i \leq j \leq i \leq n$, let $P(i, j)$ be the problem of determining $opt(i, j)$, the maximum sum of a route starting from entry $T(i, j)$.

Recurrence: if $i < n$, we have

$$opt(i, j) = T(i, j) + \max[opt(i+1, j), opt(i+1, j+1)],$$

since the best route starting at (i, j) consists of the initial entry, then follows the best route

starting from either the below-left entry of below-right entry.

Base cases: For $1 \leq j \leq n$, $opt(n, j) = T(n, j)$ since there are no entries below row n.

Order of computation: Since the solution to $P(i, j)$ depends on the solutions to $P(i+1, j)$ and $P(i+1, j+1)$, we solve subproblems by decreasing order of i and any order of j.

Time complexity: Since all routes start at the top left entry, the final answer is $opt(1, 1)$. There are $O(n^2)$ subproblems, each solved in constant time, so the overall complexity if $O(n^2)$

- Discuss each of the five headings listed above, since this way of setting out a solution is recommended by the course.

Question 3 *Drill*

UNSW is doing a fire drill. It consists of n rooms and m corridors ($m \geq n-1$), where each corridor connects two different rooms. There are x students who must be moved from room 1 to room n. Your job is to divide the class of x students into several waves. Each wave will be released from room 1, and make their way through the corridors. To prevent overcrowding, each corridor has a limit l_i, which is the maximum number of students in a single wave who can use this corridor. Once all students in this wave have reached room n, the next wave of students will be released from room 1.

Design a polynomial time algorithm which determines the minimum number of waves that must be formed.

What is the maximum number of students in a single wave?

Construct a flow network with n vertices representing the rooms. The ith corridor is represented by a pair of directed edges between the two rooms it connects, both with capacity l_i. We omit edges to vertex 1 and edges from vertex n, then denote vertex 1 as the source and vertex n as the sink. Using the Edmonds-Karp algorithm, we find a maximum flow, and let its value be y. Note that the amount of flow through each edge i is at most the capacity l_i, so any valid flow obeys the only constraint: no more than l_i students from each wave can use corridor i.

Therefore y is the largest possible size of a wave, so the number of waves is minimised by sending y students at a time. The minimum number of waves will then be $\lceil x/y \rceil$.

Constructing the graph takes $O(n+m)$ time and running the Edmonds-Karp algorithm takes $O(nm^2)$ time, so the overall time complexity is $O(nm^2)$. Since $n \leq m+1$ and the input is of length at least m, this is a polynomial time algorithm.

- Get them to explain why you should let as many people through as possible in each wave (trivial, but worth asking).
- Make sure they add edges in both directions between rooms.
- Make sure they get the ceiling function right.

Question 4 *Array*

There is a 1-indexed array A of n unknown integers ($n \geq 2$), except that you know $A[1] = 1$ and $A[n] = n - 1$. You are allowed to ask queries of the form "What is value at index i?", where you may choose any index i.

Design an algorithm which uses $O(\log n)$ queries and determines the index of an element whose value is greater than or equal to the element immediately to its right. In other words, find an index i such that $A[i] \geq A[i+1]$.

What information does querying the middle element give you?

Let $B[i] = A[i] - i$. We have $B[1] = 0$ and $B[n] = 1$. Binary search for a place where B increases, i.e. an index i where $B[i] < B[i+1]$.

More details: Initially let $l = 1$ and $r = n$. While $r - l > 1$, let $m = \lfloor (l+r)/2 \rfloor$ and get $B[m] = A[m] - m$. If $B[m] > A[l]$, set $r = m$ to recurse left, otherwise $B[m] < A[r]$, so set $l = m$ to recurse right. The invariant is that $A[l] < A[r]$. The final answer is the final value of l (which will equal $r - 1$).

- You can solve the problem without the array B, just query the middle element and explain how you determine whether to go left or right.
- Don't be too fussy about the details of the binary search.
- This problem might be pretty hard, so people will likely come for help.

Question 5 *Lizard*

There is a rectangular grid with R rows and C columns. In row r and column c, there is a stone of height h_{rc}, which holds a_{rc} lizards. Both h_{rc} and a_{rc} are non-negative integers. If h_{rc} is zero, this denotes that there is no stone at (r, c) and hence a_{rc} is guaranteed to also be zero.

Each lizard can jump between two stones if they are separated a distance of at most d. In other words, it can jump from a stone at (r_1, c_1) to a stone at (r_2, c_2) (which may be occupied by any number of lizards) if $\sqrt{(r_1 - r_2)^2 + (c_1 - c_2)^2} \leq d$.

However, the stones are not stable, so whenever a lizard leaves a stone, the height of the stone is decreased by 1. If the new height of the stone is zero, there is no more stone at at (r, c). Any remaining lizards on this stone will drown, and lizards will no longer be able to jump onto this stone.

We want to help as many lizards as possible to escape the grid. A lizard escapes if it can jump between rocks, then take a jump of at most distance d to take them beyond the boundary of the grid.

Design a polynomial time algorithm to find the maximum number of lizards that can escape from the grid.

First solve the case where initially there is only one lizard. Then generalise to allow several lizards all starting on the same stone, and finally to the full problem. Note that each initial height h_{rc} provides a constraint on the number of lizards which can use (r.e. jump from) that stone.

Construct a flow network with a (super-)source s, a (super-)sink t, and for each stone (r, c), two vertices $in_{r,c}$ and $out_{r,c}$.

- For each stone (r, c), place an edge from the source to $in_{r,c}$ with capacity $a_{r,c}$, representing the lizards starting at this stone.
- For each stone (r, c), place an edge from $in_{r,c}$ to $out_{r,c}$ with capacity $h_{r,c}$, representing

the capacity of this stone (r.e. the number of lizards which can depart it).

- For each pair of distinct stones (r, c) and (r', c') (of which there are $O((RC)^2)$), place an edge of infinite capacity from $out_{r,c}$ to $in_{r,c}$, representing any number of lizards moving from (r, c) to (r', c').
- For each stone (r, c) within d of the boundary (r.e. $r \leq d$, $(R + 1) - r \leq d$, $c \leq d$ or $(C + 1) - c \leq d$), place an edge of infinite capacity from $out_{r,c}$ to t, representing any number of lizards escaping from this stone.

Each unit of flow in this network represents one of the starting lizards jumping between stones without drowning and eventually escaping the grid, so the maximum number of lizards escaping is the size of the maximum flow. We therefore find the answer by running the Edmonds-Karp algorithm on this flow network.

There are $2rc+2 = O(RC)$ vertices and $RC+RC+O((RC)^2)+O((RC)^2) = O((RC)^2)$ edges in this flow network. Therefore constructing the graph takes $O((RC)^2)$ time, and running Edmonds-Karp takes $O((RC)^5)$ time. The length of the input is at least RC, so the algorithm runs in polynomial time.

- This problem is relatively routine, so feel free to spend a bit of time making sure they have all the details of the flow network (e.g. super source and super sink, vertex capacities, edge capacities) right.

Question 6 *Digits*

You are given a positive integer n and a decimal digit k. Your task is to count the number of n-digit numbers (without leading zeros) in which the digit k appears an even number of times. Note that we consider 0 to be an even 1-digit number.

Design a dynamic programming algorithm which solves this problem and runs in $O(n)$ time.

1 Consider deciding the number one digit at a time. What state could we use?

2 Try a dp with the following states:

- even(i), the quantity of numbers with i decimal digits where k appears an even number of times.
- odd(i), the quantity of numbers with i decimal digits where k appears an odd number of times.

All numbers with $i > 0$ digits consist of a nonzero leading digit, then $i - 1$ more digits that can be zero. We'll count how many numbers have an even number of digits k and how many have an odd number of digits k. We first deal with the case $k \neq 0$, and then we will discuss the modifications required when $k = 0$.

Subproblems: for $1 \leq i \leq n$, let $P(i)$ be the problem of determining even(i), the quantity of numbers with i decimal digits where k appears an even number of times, and odd(i), the quantity of numbers with i decimal digits where k appears an odd number of times.

Recurrence: for $1 < i < n$, we have

$$\text{even}(i) = 9 \times \text{even}(i-1) + \text{odd}(i-1)$$
$$\text{odd}(i) = 9 \times \text{odd}(i-1) + \text{even}(i-1).$$

If the leading digit is not k, then the parity of digits k in the remaining $i-1$ digits should be maintained. If instead the leading digit is k, then the parity is reversed. There are ten choices for the leading digit, namely k and the nine other digits. At the last step however, 0 is forbidden as the leading digit, so we have

$$\begin{aligned}\text{even}(n) &= 8 \times \text{even}(n-1) + \text{odd}(n-1)\\ \text{odd}(n) &= 8 \times \text{odd}(n-1) + \text{even}(n-1).\end{aligned}$$

Base cases: we have

$$\text{even}(1) = 9 \text{ and } \text{odd}(1) = 1,$$

since there is only one 1-digit number with an odd number of instances of digit k (namely k itself), and the remaining nine have no instances of k.

The solution to $P(i)$ depends on the solution to $P(i-1)$, so we solve subproblems in increasing order of i.

The final answer is simply $\text{even}(n)$, the quantity of n-digit numbers without leading zeros where digit k appears an even number of times.

There are n subproblems, each solved in constant time, so the overall time complexity is $O(n)$.

Special case: if $k = 0$, the final step of the recurrence is

$$\begin{aligned}\text{even}(i) &= 9 \times \text{even}(i-1)\\ \text{odd}(i) &= 9 \times \text{odd}(i-1)\end{aligned}$$

since the leading digit must be nonzero.

Combinatorics based solutions are not accepted, as the question asks for a DP solution.

Question 7 *Pairs*

You are given an array A of n positive integers, each at most M. For each pair of distinct indices $1 \leq i < j \leq n$, consider the corresponding sum $A[i] + A[j]$.

Design an algorithm which determines the kth largest of these sums and runs in $O(n \log n \log M)$ time.

For example, suppose $n = 4$, $k = 4$ and the array elements are $2, 5, 3, 4$. Going over pairs of distinct indices, we encounter the corresponding sums $5, 6, 7, 7, 8, 9$, so the correct answer is 7. Note that 7 appears twice in the list; it is both the third largest sum and the fourth largest sum.

For a given positive integer S, can you determine the number of pairs of indices with corresponding sum greater than or equal to S in $O(n \log n)$ time?

We know that each pair must have sum at most $2M$.

For some $0 < S \leq 2M$, we can count the number of pairs with sum at least S in $O(n \log n)$ as above. Then:

[A] If the number of such pairs is less than k, the answer must be strictly larger than S.

[B] If the number of pairs is greater than or equal to k, the answer must be smaller than or equal to S.

We want to find the largest S such that the number of pairs equal or larger than S is not less than k. The above criterion allows us to find this value of S by binary search.

This binary search requires $O(\log M)$ steps, each of which takes $O(n \log n)$, so the overall runtime is $O(n \log n \log M)$.

Note the termination condition we used; other choices might be incorrect. In particular multiple pairs could have the same size, so finding an S with *exactly* $k-1$ larger pairs won't work. This should only receive a minor penalty.

Question 8 *Tasks*

Alice has n tasks to do, the ith of which is due by the day d_i. She can work on one task each day, starting from day 1, and each task takes one day to complete. Morever, Alice is a severe procrastinator and wants to accomplish every task as close as possible to its due date. If Alice finishes the ith task on day j, her rage will increase by $d_i - j$.

Design an $O(n \log n)$ algorithm that determines whether all tasks can be completed by their deadlines, and if so, outputs the minimum total rage that Alice can accumulate.

1 Suggest that they initially just think about whether it is possible or not (ignore the rage).

2 "If $d_i < d_j$, is it true that we should always complete task i before task j". *The answer to this is that while we don't necessarily need to complete i before j, doing so will never make a solution worse (using an exchange argument proof). This strongly suggests that we should sort the tasks by d.*

Firstly, sort the tasks by their due date in descending order. Then, for each task, (greedily) choose the latest day available that isn't after the due date. This can be done by maintaining a variable for the earliest day that has a task, then assigning the task to the day before that, if it occurs before the due date, or the due date otherwise. If we attempt to assign to day 0 or earlier, then report that Alice cannot complete all tasks on time.

The total rage can be calculated either incrementally while we determine which day each task is done, or afterwards if the date each task was completed was recorded in an array.

The initial sort takes $O(n \log n)$, and the subsequent greedy assignment takes $O(n)$ (with each task taking exactly $O(1)$ time), so the overall time complexity is $O(n \log n)$.

Proof of correctness:

There are two aspects of our algorithm that we need to prove to show that it is correct:

[A] it correctly determines whether all tasks can be completed before their due date, and

[B] if so, it produces the minimum possible total rage.

If there no valid scheduling of tasks exists, then we obviously won't be able to find one. Otherwise, a valid scheduling must exist. We'll show that we can transform any valid scheduling into the one produced by our algorithm without increasing total rage.

Firstly, we can permute which task is done on which day so that if task i is scheduled before task j, then $d_i \leq d_j$. This is true because if $d_i > d_j$ for any two tasks but i is scheduled before j, then swapping those two will still give a valid schedule.

Additionally, we can further permute it so that the relative order of tasks is identical, since tasks with the same due date can be swapped without changing the total rage or validity of a schedule.

Now, our algorithm will always schedule tasks on the latest day possible under this ordering. We can then repeatedly move individual tasks back a day until we obtain the schedule produced by our algorithm. Doing so can only decrease total rage.

This means that our algorithm will always find a schedule if one exists, and the one it finds will always have minimum total rage.

Iterating over days, from the largest due date down to zero, would exceed the required time complexity. This alternative approach would take $O(n \log n + \max d_i)$, which isn't $O(n \log n)$ as d_i is unbounded.

Question 9 *Minimum*

Assume that you are given an $n \times n$ table; each cell of the table contains a distinct number. A cell is a local minimum if the number it contains is smaller than the numbers contained in all neighbouring squares which share an edge with that cell. Thus, each of the four corner squares has only two neighbours sharing an edge; $4(n-2)$ non corner squares along the edges of the table have three neighbouring squares and all internal squares have four neighbouring squares. You can only ask queries which, given numbers i and j, where $1 \leq i, j \leq n$, obtain the value in the cell (i, j).

Design an algorithm which finds a local minimum and makes only $O(n)$ many queries.

What's a naive solution to find the local minimum without explicitly querying every cell in order?

Intuition: We know that since all numbers are distinct, a local minimum has to occur within the grid. And we can get to a local minimum by following squares that are neighbours of the current cell that has a smaller value than the current cell.

Solution: Firstly, we query the middle-most 2 columns and the middle-most 2 rows and get the smallest value of all the cells that we have queried. Without loss of generality, we assume the smallest value, s, is in the bottom left quadrant. Since every number in the row above and the column to its right are bigger than s, if we were to get to the local minimum by following neighbours of s that are less than s, we know that we will never leave the bottom left quadrant and go into any other quadrant. Therefore, a local minimum has to exist in the bottom left quadrant, reducing our search space by a factor of 4.

Time complexity: Reducing an $n \times n$ search space by a factor of 4 requires $4n - 4$ queries, and $4n - 4$ comparisions to find the smallest element. Therefore, the overall time complexity is

$$(8n - 8) + \frac{8n - 8}{4} + \frac{8n - 8}{4^2} + \cdots = O(n).$$

Question 10 *Aleks*

Aleks received an offer from UNSW and he wants to graduate as soon as possible. His program requires him to complete n courses in an order of his choice. The courses are labelled $1, 2, \ldots, n$, where course i takes t_i weeks to complete. Aleks gives you these values in an array A.

However, some pairs of courses *overlap*. If courses i and j overlap, then a student who has already completed either course can complete the other in a number of weeks less than both t_i and t_j. Aleks has produced another array B with m entries. Each entry consists of an *unordered* pair of *distinct* courses which overlap (say $p = \{i, j\}$), as well as the number of weeks t_p required to complete

either course if the other has already been completed. For each such pair, you are guaranteed that $t_p < \min(t_i, t_j)$.

Design an $O((n+m)\log(n+m))$ time algorithm that finds the minimum number of weeks required to complete all n courses.

Construct a graph where each course is represented by a vertex, and each pair of overlapping courses is represented by an edge. How can you account for the times t_i and $t_{\{i,j\}}$?

For each i, the number of weeks taken to complete course i will either depend on one other course j (in which case $t_{\{i,j\}}$ weeks are required), or no course (t_i weeks). We can simplify this by introducing a new course numbered 0 that takes zero time and represents "no course", as well as n extra overlaps $t_{\{0,k\}} = t_k$ for $k = 1, \ldots, n$. This allows us to begin with course 0 and henceforth only consider times arising from an overlap, rather than the direct cost of doing any course.

Construct a graph G with $n+1$ vertices and $m+n$ undirected edges. The vertices are labelled from 0 to n, with each corresponding to a course. Each edge $e = \{i, j\}$ corresponds to a pair of overlapping courses and has weight $t_{\{i,j\}}$.

Claim: For any given order of courses, the number of weeks taken to do all courses is the total weight of some spanning tree of G.

Proof: If the number of weeks taken to complete course i depends on its overlap with course j (potentially 0), then select the edge between i and j. Let the subgraph formed by these selected edges be H. Now H has n edges, as courses 1 to n were completed. But H is also acyclic as there cannot be a cycle of dependencies; later courses depend on earlier courses. Therefore H is a spanning tree.

Clearly, the minimum number of weeks is achieved by the minimum spanning tree, which can be found using Kruskal's algorithm, implemented with the Union-Find data structure.[a]

We can also recover a valid ordering of courses from the minimum spanning tree by doing a DFS or BFS starting at node 0, though this isn't necessary to obtain the minimum time needed.

Our adjusted graph will have $m + n$ edges and $n + 1$ vertices, so it takes $O(n + m)$ time to construct an adjacency list representation of the graph. Kruskal's algorithm then finds the minimum spanning tree in $O(|E| \log |E|) = O((n + m)\log(n + m))$ time, so the overall time complexity of our algorithm is also $O((n + m)\log(n + m))$.

[a] Prim's algorithm is also acceptable, using an augmented heap.

Question 11 *Knights*

There is a $n \times n$ chess board in front of you, and an infinite supply of knights to place on it. Some squares are known to be damaged, meaning you cannot place any knights on them. The rest of the squares are undamaged, and you can place at most one knight on each undamaged square.

As per standard chess rules, two knights are said to attack each other if the distance between the two knights is $\sqrt{5}$ units. Design a polynomial time algorithm which determines the maximum number of knights that can be placed on undamaged squares such that no two knights attack each other.

Consider the graph formed by interpreting squares as vertices and interpreting pairs of squares

from which two knights can attack each other as edges. What special property does this graph have?

First, separate the grid into two parts, the ones which are white and the ones which are black. Clearly if two knights attack each other, they are in opposite parts of the grid. We need to find the maximum independent set of vertices (one in which no two vertices in the set are adjacent) of the resultant bipartite graph.

Construct a flow network with the following:

- A vertex for each undamaged square in the grid.
- An edge with capacity 1 from a white undamaged square to a black undamaged square, for each pair of undamaged squares such that two knights placed on them would attack each other.
- A supersource, from which there is an edge of capacity 1 to each white square.
- A supersink, to which there is an edge of capacity 1 from each black square.

Run the Edmonds-Karp algorithm on this graph to obtain f, the number of vertices in the minimum vertex cover.

The overall time complexity is polynomial, since constructing the graph and the Edmonds-Karp algorithm both take polynomial time.

- The proofs in the final paragraph are tricky, and the solution doesn't include them. Work with the student to fill in the details.

Question 12 *And*

You are given a non-negative integer m and a sequence A of length n, where each $0 \leq A[i] < 2^m$ for all i. Your task is to find a subsequence B of maximum length such that $B[i] \,\&\, B[i+1] \neq 0$ for all $1 \leq i < n$, where & denotes bitwise AND.

Design a dynamic programming algorithm which solves this problem and runs in $O(mn)$ time.

1 First, let them know that they should begin by trying to find the length of the longest sequence. Once they have a method to find the length of the longest sequence, then they should try to find the sequence itself.

2 Ask them to solve it in $O(n^2m)$. *The solution is to have $dp(i)$ be the longest subarray ending at index i. Note that $dp(i) = \max_{j<i \text{ where } B[i]\&B[j] \neq 0} dp(j) + 1$, which can be found in $O(nm)$, giving an overall solution of $O(n^2m)$.*

3 Once they have solved the problem in $O(n^2m)$, suggest that they try a solution with $O(nm)$ states.

Note first that each $A[i]$ can be written as an m-bit number, potentially with leading zeros.

Subproblems: For $1 \leq i \leq n$, let $P(i)$ be the problem of determining $\text{opt}(i)$, the length of the longest valid subsequence of $A[1..i]$ ending at $A[i]$ and $\text{pred}(i)$, the index in A of the penultimate entry in such a subsequence. Also, for $1 \leq i \leq n$ and $1 \leq j \leq m$, let $Q(i,j)$ be the problem of determining $f(i,j)$, the length of the longest valid subsequence of $A[1..i]$ where the jth bit[a] of the last selected element is 1, and $g(i,j)$, the index in A of the last entry in such a subsequence.

Recurrence: For $i > 1$, let

$$j^* = \arg\max_j \{f(i-1, j) \mid \text{the } j\text{th bit of } A[i] \text{ is } 1\}.$$

If $A[i]$ is the last entry of a valid subsequence, it must have a bit in common with the penultimate entry of the subsequence. There is no other constraint, so we pick the bit j^* which allows $A[i]$ to extend a valid subsequence of maximum length. Then

$$\text{opt}(i) = f(i-1, j^*) + 1$$

and

$$\text{pred}(i) = g(i-1, j^*).$$

Note that if $A[i]$ is zero, then j^* is undefined, so we set $\text{opt}(i) = 1$ and $\text{pred}(i)$ is undefined.

Also, for $i > 1$ and all $1 \leq j \leq m$,

$$f(i,j) = \begin{cases} \text{opt}(i) & \text{if the } j\text{th bit of } A[i] \text{ is } 1 \\ f(i-1,j) & \text{if the } j\text{th bit of } A[i] \text{ is } 0 \end{cases}$$

with $g(i, j) = i$ in the first case and $g(i, j) = g(i - 1, j)$ in the second case. Clearly, if $A[i]$ has a 0 in bit j, then we should refer to the solution of $Q(i, j)$. However, if $A[i]$ has a 1 in bit j, it is in fact the best end index available. In this case, any solution to $Q(i, j)$ ending at $A[k]$ where $k < i$ is suboptimal, since appending $A[i]$ yields a subsequence which is:

- longer by one,
- valid, since $A[k]$ and $A[i]$ both have 1 in the jth bit, so $A[k] \,\&\, A[i]$ is nonzero, and
- also has a 1 in the jth bit of its last entry.

Therefore we simply take the longest subsequence ending at $A[i]$, which is solved in $P(i)$.

Base cases: We have $\text{opt}(1) = 1$ and $\text{pred}(1)$ undefined, since the first entry alone is a valid subsequence with no penultimate element. For $1 \leq j \leq m$,

$$f(1,j) = \begin{cases} 1 & \text{if the } j\text{th bit of } A[1] \text{ is } 1 \\ 0 & \text{if the } j\text{th bit of } A[1] \text{ is } 0 \end{cases}$$

with $g(i, j) = 1$ in the first case and undefined otherwise, since the only candidate subsequences are either the sole element $A[1]$ or the empty subsequence.

Since the solution to $P(i)$ depends on the solutions to $Q(i-1, \cdot)$ *and* the solution to $Q(i, j)$ depends on the solutions to $P(i)$ and $Q(i-1, j)$, we solve subproblems by increasing order of i, at each stage solving $P(i)$ and then all $Q(i, j)$ in any order of j. The overall longest valid subsequence must end at some entry $A[i]$, so its length is given by

$$\max_{1 \leq i \leq n} \text{opt}(i).$$

The index of the last entry in such a subsequence is

$$i^* = \arg\max_{i:1 \leq i \leq n} \text{opt}(i),$$

and the rest of the subsequence is found by backtracking; the second last entry is $\text{pred}(i^*)$, the third last is $\text{pred}(\text{pred}(i^*))$, and so on.

There are n subproblems $P(i)$, each solved in $O(m)$ time, and nm subproblems $Q(i, j)$, each solved in constant time, so the overall complexity is $O(nm)$.

[a] "jth bit" means jth least significant bit throughout this solution.

Note that to fully solve the problem they need to find the subsequence, not just its length. However, finding the length is about 90% of the problem, so if they solve that they are very close.

Also, watch out for incorrect solutions which inadvertently only find subarrays where all elements have a bit in common.

Question 13 *Bits*

Given a 32 bit unsigned number, count the number of 1 bits it contains in no more than 20 operations. You are allowed to use any arithmetic operation (addition, subtraction, multiplication, division, modulus) and any logical operation provided in C (bit shift, AND, OR, NOT, XOR).

You are allowed to use variable assignment to store partial calculations, which is not counted as an operation. However, you cannot use the following: control flow (e.g. if, for, while), functions, arrays, typecasting, a variable called potato.

The only operations you need are addition, bit shift and AND.

This is a divide and conquer problem, how can we solve the problem in batches?

Let the ith bit of A be denoted as A_i. Then we want

$$ans = A_0 + A_1 + \cdots + A_{63}.$$

Applying divide and conquer, we want

$$ans = B_0 + B_1 + \cdots + B_{31}.$$

where $B_0 = A_0 + A_1$, $B_1 = A_2 + A_3$, $\cdots$ and $B_{31} = A_{62} + A_{63}$

Doing this through code looks like

```
B = ((A & 0b10101010101010101010101010101010) >> 1) +
    (A & 0b01010101010101010101010101010101);
```

Now every 2 bit chunk of B stores the bitcount of the corresponding 2 bit chunk of A.

Similarly, we want

$$ans = C_0 + C_1 + \cdots + C_{15}$$

where $C_0 = B_0 + B_1$, $C_1 = B_2 + B_3$, $\cdots$ and $C_{15} = B_{30} + B_{31}$

Doing this through code looks like

```
C = ((B & 0b11001100110011001100110011001100) >> 2) +
    (B & 0b00110011001100110011001100110011);
```

Hence, for a total of 20 operation, a solution is:

```
B = ((A & 0b10101010101010101010101010101010) >> 1) +
     (A & 0b01010101010101010101010101010101);

C = ((B & 0b11001100110011001100110011001100) >> 2) +
     (B & 0b00110011001100110011001100110011);

D = ((C & 0b11110000111100001111000011110000) >> 4) +
     (C & 0b00001111000011110000111100001111);

E = ((D & 0b11111111000000001111111100000000) >> 8) +
     (D & 0b00000000111111110000000011111111);

F = ((E & 0b11111111111111110000000000000000) >> 16) +
     (E & 0b00000000000000001111111111111111);
```

Note that students should demonstrate this concept but they do not need to provide this code exactly.

Further optimisations: The AND on the second last line is redundant, removing this takes us to 19 operations.

Exercise to the reader - use only 14 queries!