Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

Section 1: Hashing, Streaming

1 Open Addressing

Recall that in hashing with chaining, each table cell stores a bucket of keys. In *open addressing*, all keys are stored directly inside the table. When a collision occurs, we try another location according to some probe sequence.

Suppose we have a table of size $m = 11$, hash function

$$h(x) = x \bmod 11,$$

and linear probing rule

$$h_t(x) = (h(x) + t) \bmod 11$$

for $t = 0, 1, 2, \ldots$.

This means that the key x first tries slot $h_0(x) = h(x)$. If that slot does not answer the question, the algorithm tries $h_1(x)$, then $h_2(x)$, and so on. The sequence

$$h_0(x), h_1(x), h_2(x), \ldots$$

is called the *probe sequence* for x.

For this problem, assume there are no deletions. The operations work as follows:

- `put(x)` follows the probe sequence for x until it finds an empty slot, then stores x there. If it sees x along the way, it can stop because x is already present.
- `find(x)` follows the probe sequence for x until either it finds x, in which case it returns `True`, or it reaches an empty slot, in which case it returns `False`. Reaching an empty slot proves x is not present, because `put(x)` would have inserted x there before probing any later slots.

The current table is:

Index	0	1	2	3	4	5	6	7	8	9	10
T		12	23		15	5		18	29		21

(a) Draw the probe sequence used to insert 34, and show the updated table.

(b) Starting from the updated table, draw the probe sequence used to search for 56. Does the search succeed or fail?

(c) The table above was built using linear probing, so nearby collisions can form clusters. To analyze the probabilistic behavior more cleanly, suppose instead that each new key has a uniformly random probe sequence over the table locations. If the load factor is $\alpha = n/m$, what is the probability that the first probe collides? Under the simplifying assumption that each probe is an independent uniformly random location, what is the expected number of probes needed to insert a new key?

(d) Explain why open addressing becomes slow as α approaches 1, and briefly compare this with chaining.

Solution:

(a) We have

$$h(34) = 34 \bmod 11 = 1.$$

Linear probing checks indices

$$1 \to 2 \to 3.$$

Indices 1 and 2 are occupied by 12 and 23, so the first empty slot is index 3. After inserting 34, the table becomes:

Index	0	1	2	3	4	5	6	7	8	9	10
T		12	23	34	15	5		18	29		21

(b) We have

$$h(56) = 56 \bmod 11 = 1.$$

Starting from the updated table, linear probing checks

$$1 \to 2 \to 3 \to 4 \to 5 \to 6.$$

The keys at indices $1, 2, 3, 4, 5$ are not 56, and index 6 is empty. Therefore 56 is not in the table, so the search fails after 6 probes.

(c) If the load factor is $\alpha = n/m$, then an α fraction of the table is occupied. A uniformly random first probe therefore collides with probability

$$\alpha.$$

Under the independent-probe model, each probe hits an empty location with probability $1 - \alpha$. Thus the number of probes until insertion succeeds is a geometric random variable with success probability $1 - \alpha$, so the expected number of probes is

$$\frac{1}{1-\alpha}.$$

For the original table, $n = 7$ and $m = 11$, so $\alpha = 7/11$. The idealized expected insertion cost would be

$$\frac{1}{1 - 7/11} = \frac{11}{4}.$$

For the updated table, $\alpha = 8/11$, so the idealized expected insertion cost would be

$$\frac{1}{1 - 8/11} = \frac{11}{3}.$$

(d) As α approaches 1, the probability of hitting an empty slot, $1 - \alpha$, approaches 0. Therefore the expected number of probes, $1/(1-\alpha)$, grows very large. This is why open-addressing hash tables are usually resized before they become close to full.

In chaining, collisions create longer buckets, but the table can still store multiple keys at the same hash location. In open addressing, a collision forces the key to search for another array slot. This can be very fast when the table is sparse, but it becomes fragile when the table is crowded or when the probe rule creates clusters.

2 [Bonus] Perfect Hashing

You are given a static set S of n keys from a universe $[U]$. Your goal is to preprocess S into a dictionary data structure that supports membership queries.

Design a randomized hashing-based data structure with the following guarantees:

- $O(n)$ expected preprocessing time,
- $O(n)$ space,
- $O(1)$ worst-case query time after preprocessing.

Your answer should describe the data structure, the preprocessing algorithm, the query algorithm, and why the stated time and space guarantees hold.

Solution: Use two-level perfect hashing.

First, choose a hash function $h : [U] \to [n]$ from a universal hash family. Let B_i be the set of keys that hash to bucket i, and let $b_i = |B_i|$. Repeatedly sample h until

$$\sum_{i=0}^{n-1} b_i^2 = O(n).$$

Such an h exists with constant probability. In particular, for a universal hash family,

$$\mathbb{E}\left[\sum_i b_i^2\right] = O(n),$$

so by Markov's inequality, a constant number of trials in expectation suffices to find a top-level hash function whose buckets have total squared size $O(n)$.

For each bucket i, allocate a second-level table of size b_i^2. Then choose a separate hash function

$$h_i : [U] \to [b_i^2]$$

from a universal hash family until no two keys in B_i collide under h_i. For a fixed bucket with b_i keys, the expected number of colliding pairs under a random h_i is at most

$$\binom{b_i}{2} \cdot \frac{1}{b_i^2} < \frac{1}{2}.$$

Thus a collision-free h_i is found after $O(1)$ trials in expectation.

The total space is

$$\sum_i O(b_i^2) = O(n).$$

The expected preprocessing time is also $O(n)$, since the top-level hash function takes expected $O(1)$ trials and each second-level table takes expected linear time in its allocated size.

To answer a query for key x, compute $i = h(x)$, then compute $h_i(x)$ and check the corresponding slot in bucket i's second-level table. Since each second-level table is collision-free for the stored keys, this takes $O(1)$ worst-case time.

3 Streaming Algebra

You are given a continuous data stream of N integers, $A = (a_1, a_2, \ldots, a_N)$. You do not know N in advance, but you are guaranteed that $N \geq 4$. You may read the stream exactly once, and you may not store the array.

Your goal is to compute the maximum possible value of

$$a_i - 2a_j + 3a_k - 4a_l$$

over all choices of indices satisfying

$$1 \leq i < j < k < l \leq N.$$

Assume you are restricted to $O(\log S)$ bits of memory, where S is the maximum possible absolute value of the final expression.

Design a deterministic streaming algorithm for this problem. Your answer should state the variables your algorithm maintains, how those variables are updated when a new stream element arrives, why the algorithm is correct, and why it uses only $O(\log S)$ bits of memory.

Solution: Maintain four variables, one for each prefix of the expression:

$$\begin{aligned} S_1 &= \max a_i, \\ S_2 &= \max(a_i - 2a_j), \\ S_3 &= \max(a_i - 2a_j + 3a_k), \\ S_4 &= \max(a_i - 2a_j + 3a_k - 4a_l). \end{aligned}$$

Initially, set

$$S_1 = S_2 = S_3 = S_4 = -\infty.$$

Equivalently, $-\infty$ can be replaced by any integer smaller than every possible value these expressions could take.

When a new stream element x arrives, perform the updates in the following strict reverse order:

$$\begin{aligned} S_4 &\leftarrow \max(S_4, S_3 - 4x), \\ S_3 &\leftarrow \max(S_3, S_2 + 3x), \\ S_2 &\leftarrow \max(S_2, S_1 - 2x), \\ S_1 &\leftarrow \max(S_1, x). \end{aligned}$$

At the end of the stream, output S_4.

For example, suppose the stream is

$$(4, 1, 5, 2).$$

After reading 4, only S_1 becomes finite:

$$(S_1, S_2, S_3, S_4) = (4, -\infty, -\infty, -\infty).$$

After reading 1:

$$S_2 = 4 - 2(1) = 2,$$

so

$$(S_1, S_2, S_3, S_4) = (4, 2, -\infty, -\infty).$$

After reading 5:

$$S_3 = 2 + 3(5) = 17,$$

so

$$(S_1, S_2, S_3, S_4) = (5, 2, 17, -\infty).$$

After reading 2:

$$S_4 = 17 - 4(2) = 9.$$

Thus the algorithm outputs 9, which corresponds to choosing $i = 1, j = 2, k = 3, l = 4$:

$$4 - 2(1) + 3(5) - 4(2) = 9.$$

The reverse update order is what enforces the strict inequalities.

When the current stream element x is considered as the final term a_l, the algorithm computes $S_3 - 4x$ before updating S_3 with x. Therefore, the S_3 used in this computation was built only from elements that arrived before x. So the indices inside S_3 are all strictly smaller than the index of x.

The same reasoning applies recursively. When x is considered as a_k, the algorithm uses the old value of S_2; when x is considered as a_j, it uses the old value of S_1. Since no state can use the current element more than once during the same update step, the selected indices must satisfy $i < j < k < l$.

The algorithm stores only four integers, each with $O(\log S)$ bits, so the total memory usage is $O(\log S)$ bits.

Section 2: Graphs, Greedy Algorithms

4 Remove Covered Intervals

Given an array of intervals, remove all intervals that are covered by another interval in the list.

The interval $[a, b]$ is covered by the interval $[c, d]$ if and only if $c \leq a$ and $b \leq d$.

Find the number of remaining intervals.

Example: $[[1, 5], [5, 20], [3, 12], [4, 12]]$ In this example, the interval $[3, 12]$ fully covers the interval $[4, 12]$. Hence, the remaining intervals would be $[[1, 5], [5, 20], [3, 12]]$.

Solution:

Sort the intervals in increasing order of start time, and tie-break in decreasing order of end time. For example, the interval $[1, 6]$ would appear before the interval $[1, 4]$.

Maintain the largest interval end you have seen so far as you process each interval (call this variable `highest`). For each interval $[a, b]$, we have 2 cases.

1. If $b \leq$ `highest`, that means there exists some interval that started before `a` (since we ordered by start time), which also ends after `b`. Hence, we can ignore this interval.
2. If $b >$ `highest`, set `highest` to b, and keep the interval.

Return the number of intervals that remain at the end.

In the given example, we will perform the following steps:

1. Sort the intervals: [[1, 5], [3, 12], [4, 12], [5, 20]]
2. Process [1, 5]: Assign `highest` to 5 and keep the interval
3. Process [3, 12]: `highest` < 12; Assign `highest` to 12 and keep the interval
4. Process [4, 12]: `highest` $= 12$; Remove the interval
5. Process [5, 20]: `highest` < 20; Assign `highest` to 20 and keep the interval
6. Our final solution will retain exactly 3 intervals.

5 Money Madness

Anirudh is on vacation but forgot to convert his US dollars (USD) to Indian Rupees (INR)! He visits a currency exchange booth at the airport. The booth provides an exchange rate to convert between various different currencies. Unfortunately, the exchange rate provided is never better than the actual conversion rate (you will never make money by exchanging).

(a) Given a list of exchange rates, provide an algorithm to find the maximum amount of INR he can get for 1 USD.

(b) Now, assume the booth charges a percentage fee (varies depending on currency) for each exchange. For example, if 1 USD is worth 100 INR, the currency exchange might only give you 90 INR per dollar. We represent this as an exchange rate of 0.9, because you retain 90% of the value. Once again, all exchange rates are in the range $(0, 1)$ exclusive (so you cannot profit).

Design an algorithm that finds the best sequence of exchanges for converting USD to INR (using any number of exchanges), retaining as much value as possible.

Solution:

(a) We will have to convert the given information into a graph, and utilize an Shortest Paths Tree algorithm to find the best exchange rate.

Graph Creation:

- Each vertex represents a currency.
- Each edge represents an exchange rate provided by the booth (which may differ from the real conversion rate).

An ideal algorithm for this problem would be to use Dijkstra's. However, Dijkstra's requires you to know "which vertex is closer", as this becomes the next visited vertex. In this case, it is not possible to compare between two amounts of different currencies because the real conversion rate between the two is not provided (you only know the conversion the booth offers). Hence, we will have to use a variation of Bellman-Ford.

Algorithm Modification:

- Instead of adding distances, we will multiply by the provided exchange rate.
- Instead of maintaining the smallest distance to each vertex, we will store the highest amount we can reach of each currency.

We know this works because there are no negative cycles as no conversion ever makes us money. Simply return the values stored for the goal currency at the end.

(b) We will create a graph similarly to part (a). In this subpart, we have a uniform way of comparing values at two different currencies (retaining 95% is closer to the start than retaining only 80%). In addition, a conversion still always puts us farther from our original value (we always lose money). Hence, we can use Dijkstra's instead of Bellman-Ford. We have two possible solutions here:

1. Modify only the graph: Change each edge weight `w` to be `-log(w)`. Run Dijkstra's. Return 2^{-d}, where `d` represents the distance from USD to INR in the adjusted graph.
2. Run Dijkstra's using the original graph, but multiply weights instead of adding. Also, replace the min-heap with a max-heap (visit vertices with the most amount retained). Return the distance to INR at the end.

Section 3: Linear Programming and Zero-Sum Games

6 Review Planning

You have T hours left to study for an oral exam. There are m review activities you can choose from, such as redoing a homework problem, reading lecture notes, going through a discussion worksheet, or explaining a proof to a friend.

There are k topics on the exam. Spending one hour on activity j gives a_{ij} units of preparation for topic i. You may spend a fractional number of hours on each activity. Let $x_j \geq 0$ be the number of hours that you spend on activity j.

You expect the examiner to look for areas where you are least prepared, so you want to make your study plan balanced: your goal is to maximize your weakest topic preparation, which is given by

$$\min_{i \in [k]} \sum_{j=1}^{m} a_{ij} x_j.$$

(a) Formulate this study planning problem as a linear program.

(b) Write down the dual of your LP from part (a). Briefly describe an interpretation of the dual from the examiner's perspective.

Solution:

(a) The LP is given by:

- Variables: x_j for each activity j, and variable z representing the weakest topic preparation.
- Constraints:

$$z - \sum_j a_{ij} x_j \leq 0 \text{ for each } i$$
$$\sum_j x_j = T$$
$$x_j \geq 0 \text{ for each } j$$

- Objective: Maximize z

(b) The dual LP is given by:

- Variables: w_i for each topic i, and variable y corresponding to the time constraint.
- Constraints:

$$y - \sum_i a_{ij} w_i \geq 0 \text{ for each } j$$
$$\sum_i w_i = 1$$
$$w_i \geq 0 \text{ for each } i$$

- Objective: Minimize $T \cdot y$

We can interpret w_i as a probability distribution over topics, representing the examiner's emphasis on each topic. The dual asks the examiner to choose these weights so that even the best study activity has as little weighted preparation value as possible.

7 Penalty Kicks

A kicker and a goalie are playing a zero-sum game. The kicker chooses where to shoot, the goalie chooses where to dive, and the payoff is the probability that the kicker scores. The kicker wants to maximize this probability, while the goalie wants to minimize it.

The payoff matrix is

	Dive Left	Stay Center	Dive Right
Shoot Left	0.2	0.7	0.8
Shoot Center	0.4	0.5	0.4
Shoot Right	0.8	0.7	0.2

(a) Eliminate any dominated strategies in order to reduce this 3×3 game to a 2×2 game. Justify each strategy you eliminate.

Hint: One strategy may be dominated by a mixture of other strategies.

(b) Solve the resulting 2×2 zero-sum game. What is the kicker's optimal mixed strategy, what is the goalie's optimal mixed strategy, and what is the value of the game?

(c) Write a linear program for the kicker's optimal mixed strategy in the original 3×3 game, and give the optimal solution to this linear program.

(d) Suppose a different kicker is better at shooting left, and has the payoff matrix

	Dive Left	Stay Center	Dive Right
Shoot Left	0.3	0.8	0.9
Shoot Center	0.4	0.5	0.4
Shoot Right	0.8	0.7	0.2

Would the solution to the zero-sum game still involve a mixed strategy, or should the kicker now always shoot left? Explain in one or two sentences.

Solution:

(a) Shoot Center is dominated by mixing Shoot Left and Shoot Right equally, which gives payoffs

$$(0.5, 0.7, 0.5),$$

compared to $(0.4, 0.5, 0.4)$ from Shoot Center.

After removing Shoot Center, Stay Center is dominated for the goalie by mixing Dive Left and Dive Right equally, which gives scoring probabilities $(0.5, 0.5)$ instead of $(0.7, 0.7)$. The reduced game is

	Dive Left	Dive Right
Shoot Left	0.2	0.8
Shoot Right	0.8	0.2

(b) Let p be the probability that the kicker shoots left. The expected payoff against Dive Left is $0.2p + 0.8(1-p)$, and the expected payoff against Dive Right is $0.8p + 0.2(1-p)$. Setting these equal gives $p = 1/2$, which guarantees payoff 0.5.

This is optimal for the kicker: if $p > 1/2$, the goalie can always dive left and make the scoring probability less than $1/2$, while if $p < 1/2$, the goalie can always dive right and make the scoring probability less than $1/2$. Similarly, the goalie's optimal strategy is to dive left and right with probability $1/2$ each. The value of the game is 0.5.

(c) The kicker's LP is

$$\begin{aligned} & & \text{maximize} \quad & v \\ \text{subject to} \quad & 0.2x_L + 0.4x_C + 0.8x_R \geq v \\ & 0.7x_L + 0.5x_C + 0.7x_R \geq v \\ & 0.8x_L + 0.4x_C + 0.2x_R \geq v \\ & x_L + x_C + x_R = 1 \\ & x_L, x_C, x_R \geq 0. \end{aligned}$$

One optimal solution is

$$x_L = \frac{1}{2}, \qquad x_C = 0, \qquad x_R = \frac{1}{2}, \qquad v = \frac{1}{2}.$$

(d) The solution still involves a mixed strategy. Always shooting left only guarantees scoring probability 0.3, since the goalie can dive left, while mixing equally between Shoot Left and Shoot Right guarantees scoring probability 0.55.

Section 4: NP-Completeness

8 Bad Reductions

In each part we make a wrong claim about some reduction. Explain for each one why the claim is wrong.

(a) The shortest simple path problem with non-negative edge weights can be reduced to the longest simple path problem by just negating the weights of all edges. There is an efficient algorithm for the shortest simple path problem with non-negative edge weights, so there is also an efficient algorithm for the longest path problem.

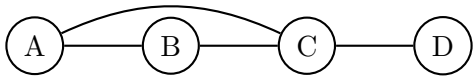
(b) Undirected Rudrata Path can be reduced to Longest Path in a DAG in the following way: Given the undirected graph G, we will use DFS to find a traversal of G and assign directions to all the edges in G based on this traversal. In other words, the edges will point in the same direction they were traversed and back edges will be omitted, giving us a DAG. If the longest path in this DAG has $|V| - 1$ edges, then there must be a Rudrata path in G, as any simple path with $|V| - 1$ edges must visit every vertex.

Solution:

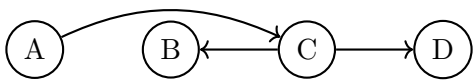
(a) The reduction is in the wrong direction. By reducing shortest simple path to longest simple path, we showed that longest simple path is at least as hard as shortest simple path, but the longest simple path problem could be much harder (indeed, we don't know of any algorithm taking less than exponential time).

(b) It is true that if the longest path in the DAG has length $|V| - 1$, then there is a Rudrata path in G. However, to prove a reduction correct, **you have to prove both directions**. That is, if you have reduced problem A to problem B by transforming instance I to instance I', then you should prove that I has a solution **if and only if** I' has a solution. In the above "reduction," one direction does not hold. Specifically, if G has a Rudrata path, then the DAG that we produce does not necessarily have a path of length $|V| - 1$–it depends on how we choose directions for the edges.

For a concrete counterexample, consider the following graph:



It is possible that when traversing this graph by DFS, node C will be encountered before node B, and thus the DAG produced will be



which does not have a path of length 3, even though the original graph did have a Rudrata path.

9 Exact 4-SAT

The Exact 4-SAT problem is defined as follows.

Input: n boolean variables $\{x_1, \ldots, x_n\}$ and clauses $\{C_1, \ldots, C_m\}$ with each containing exactly {four distinct} literals. For example, the following is an instance of Exact 4-SAT,

$$(x_1 \vee x_2 \vee \overline{x_4} \vee \overline{x_5}) \wedge (x_3 \vee \overline{x_4} \vee \overline{x_1} \vee x_2) \wedge (x_1 \vee x_3 \vee x_4 \vee x_5)$$

Note that all the 4 literals within a clause have to be distinct.

Goal: Find an assignment to the variables $x_1, \ldots, x_n$ that satisfies all the clauses.

(a) Give a polynomial time reduction from 3-SAT to Exact 4-SAT.

(b) Give a polynomial time reduction from Exact 4-SAT to 3-SAT.

(c) What does this tell us about the complexity of Exact 4-SAT?

Solution:

(a) Given a 3-SAT instance, we want to find a way to convert it into a 4-SAT instance. We can do this by creating a single additional variable y and adding it to every clause. For example, for the 3-SAT clause, $(x_1 \vee x_2 \vee \overline{x_4})$, we create the 4-SAT clause $(x_1 \vee x_2 \vee \overline{x_4} \vee y)$. However, we cannot just stop here because calling the 4-SAT solver on this instance we've constructed will just set y to be true, and all clauses in the 4-SAT instance will be satisfied, even if the original 3-SAT instance wasn't satisfiable.

So, instead, we need to create two 4-SAT clauses for each 3-SAT clause, one with y added and the other with $\overline{y}$ added. For example, for the 3-SAT clause, $(x_1 \vee x_2 \vee \overline{x_4})$,we create the two 4-SAT clauses $(x_1 \vee x_2 \vee \overline{x_4} \vee y) \wedge (x_1 \vee x_2 \vee \overline{x_4} \vee \overline{y})$. We pass these clauses to the EXACT 4-SAT solver.

Now, if the solver sets y to be true, then half of the clauses containing y are immediately satisfied. Then, the other half containing $\overline{y} = \mathsf{FALSE}$ becomes equivalent to the input to the 3-SAT instance, so the assignment of x_i values must satisfy that input. This also happens if the solver sets y to be false.

(b) To go from Exact 4-SAT to 3-SAT, we want to convert each clause with 4 literals to clauses with at most 3 literals. To do this, for each 4-SAT clause, for example $(x_1 \vee x_2 \vee x_3 \vee x_4)$, create a new variable y. Then, add the corresponding clauses $(x_1 \vee x_2 \vee y) \wedge (x_3 \vee x_4 \vee \overline{y})$ to our 3-SAT instance.

Now, if the 4-SAT instance is satisfiable, then our $3-SAT$ instance will be satisfiable by setting y to true if the first two literals in that clause are false, and setting y to false otherwise.

Similarly, if such a 3-SAT instance is satisfiable, then since either y or $\overline{y}$ will be false, one of the literals in the original 4-SAT clause has to be true. Hence, the 4-SAT instance is also satisfiable.

(c) From part (a), we showed that Exact 4-SAT is NP-Hard by constructing a reduction from 3-SAT to Exact 4-SAT. Since a satisfying assignment can be verified in polynomial time, Exact 4-SAT is also in NP, and therefore NP-complete. Additionally, part (b) shows a reduction from Exact 4-SAT to 3-SAT, implying that the two problems are polynomial-time equivalent.

10 Dominating Set

A dominating set of a graph $G = (V, E)$ is a subset S of V, such that every vertex not in S is a neighbor of at least one vertex in S. Let the Dominating Set problem be the task of determining whether there is a dominating set of size $\leq k$. We will show that the Dominating Set problem is NP-Complete. You may assume that G is connected.

(a) Show that Dominating Set is in NP.

(b) Show that Dominating Set is NP-Hard.
Hint: Try reducing from Vertex Cover or Set Cover.

Solution:

Proof that Dominating Set is in NP.
Given a possible solution, we can check that it is a solution by iterating through the vertices not in the solution subset D and checking if it has a neighbor in D by iterating through the adjacent edges. This would take at most E time because you would at most check every edge twice. We should also check that the number of vertices in D is less than k, which would take at most V, as k should be less than E. So, we can verify in $O(E)$ time, which is polynomial.

Proof that Minimum Dominating Set is NP-Hard.

Reduction from Vertex Cover to Dominating Set
The Vertex Cover problem is to find a vertex cover (a subset of the vertices) which is of size $\leq k$, where all edges have an endpoint in the vertex cover.

Suppose $G(V, E)$ is an instance of the Vertex Cover problem and we are trying to find a vertex cover of size $\leq k$. For every edge (u, v), we will add a new vertex c and two edges (c, u), and (c, v). We will pass in the same k to the dominating set. This is a polynomial reduction because we are adding $|E|$ vertices and $2|E|$ edges.

Proof of Correctness of Reduction
1. **If vertex cover has a solution, then dominating set that corresponds to it has a solution.**
Suppose we have some vertex cover of size k. By the definition of a vertex cover, every edge has either one or both endpoints in the vertex cover. Thus, if we say that the vertex cover is a dominating set, every original vertex must either be in the dominating set or adjacent to it. Now we have to figure out whether the vertices we added for every edge are covered. Since every edge has to have one or both endpoints in the vertex cover, the added vertices must be adjacent to at least one vertex in the vertex cover. Thus the vertex cover maps directly to a dominating set in the transformed problem.

2. **If dominating set in the transformed format has a solution, then the corresponding vertex cover has a solution.**
Suppose we have some dominating set of size k of the transformed format. Vertices in the dominating set either must come from the original vertices or the vertices we added. If they don't come from the vertices we added in the transformation, then from the logic stated in the previous direction, the dominating set corresponds directly to the vertex cover. If some vertices come from the transformation, then we can substitute the vertex for either of the endpoints of the edge it corresponds to without changing the size of the dominating set. Thus, we can come up with a dominating set of size k that only uses vertices from our original problem, which will directly match to a vertex cover of size k in

the original problem.

Alternative Proof that Dominating Set is NP-Hard.

Reduction from Set Cover to Dominating Set
Set Cover is to find a set cover (a subset of all the sets) which is of size $\leq k$, where all elements are covered by at least one set of the set cover.

Suppose (S, U) is an instance of set cover where U denotes the set of all distinct elements and S is a set of subsets S_i of U. We will construct a graph $G = (V, E)$ as follows. For each element u in U, construct a vertex; we will call these "element vertices". For each possible S_i construct a vertex; we will call these "set vertices". Connect each vertex S_i to all u in S_i.

Notice that if we were to run dominating set on the graph right now, we would be able to cover all the element vertices with any valid set cover, but we would have to pick every single set vertex in order to ensure that all set vertices are covered. To rectify this, connect every set vertex to every other set vertex, forming a clique. This ensures that we can cover all the set vertices by picking just one. This way, we really only need to worry about covering the element vertices.

Proof of Correctness of Reduction
1. **If set cover has a solution, then dominating set that corresponds to it has a solution.**
Suppose we have some set cover of size k. This will correspond to a dominating set of size k as well. For each set in our set cover, pick the corresponding set vertex. It follows directly from the construction of the graph and the definition of a set cover that all set and element vertices are covered.

2. **If dominating set in the transformed format has a solution, then the corresponding set cover has a solution.**
Suppose we have some dominating set D of size k. We can build a set cover C as follows: for every "set vertex" $S_i \in D$, add S_i to C and for every "element vertex" $u \in D$, pick any set S_i containing u and add S_i to C. Then $|C| \leq |D| \leq k$.

To check that C covers U, every vertex $u \in U$ must either be in D itself and added to C or has a neighboring set vertex S_j that is in D where $u \in S_j \in C$.

Since this problem is in NP and is NP-Hard, it must be NP-Complete.

11 Independent Set Approximation

In the Max Independent Set problem, we are given a graph $G = (V, E)$ and asked to find the largest set $V' \subseteq V$ such that no two vertices in V' share an edge in E.

Given an undirected graph $G = (V, E)$ in which each node has degree $\leq d$, give an efficient algorithm that finds an independent set whose size is at least $1/(d+1)$ times that of the largest independent set. Describe your algorithm and prove that the it finds an independent set of size is at least $1/(d+1)$ times the largest possible solution. Your algorithm should run in time $O(|V| \cdot |E|)$ (or less).

Solution: Initially, let G be the original graph and $I = \emptyset$. Repeat the process below until $G = \emptyset$:

1. Pick an arbitrary node v in G and let $I = I \cup \{v\}$.
2. Delete v and all its neighbors from the graph.
3. Let G be the new graph.

Let I be the independent set constructed by the greedy algorithm. At each step, I grows by one vertex and we delete at most $d+1$ vertices from the graph (since v has at most d neighbors). Hence there are at least $|V|/(d+1)$ iterations or vertices in the set. Let OPT be the size of the maximum independent set. Since $OPT \leq |V|$, we can use the previous argument to get:

$$|I| \geq \frac{|V|}{d+1} \geq \frac{OPT}{d+1}$$

This shows that we have found an independent set that is at least $1/(d+1)$ times the optimal solution.

Section 5: Divide & Conquer, FFT, Parallelism, Dynamic Programming

12 Sorted Array

Given a sorted array A of n (possibly negative) distinct integers, you want to find out whether there is an index i for which $A[i] = i$. Devise a divide-and-conquer algorithm that runs in $O(\log n)$ time.

Solution:

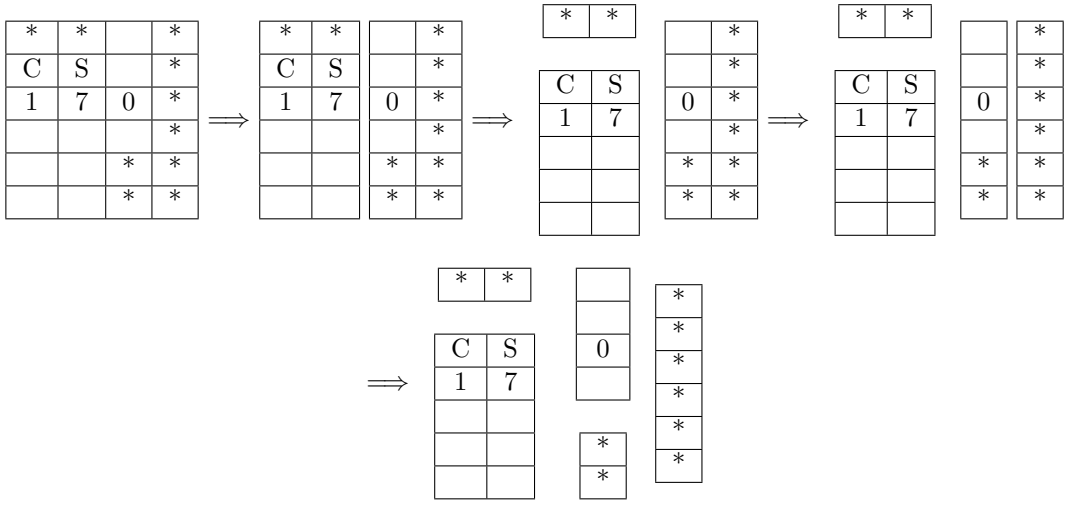
Along the same lines as binary search, start by examining $A[\frac{n}{2}]$. There are three cases for this particular index; $A[\frac{n}{2}] = \frac{n}{2}$, $A[\frac{n}{2}] > \frac{n}{2}$, and $A[\frac{n}{2}] < \frac{n}{2}$.
If $A[\frac{n}{2}] = \frac{n}{2}$, then we have a satisfactory index so we can return true.
If $A[\frac{n}{2}] > \frac{n}{2}$, then no element in the second half of the array can possibly satisfy the condition because each integer is at least one greater than the previous integer (because of the distinct property), and hence the difference of $A[\frac{n}{2}] - \frac{n}{2}$ cannot decrease by continuing through the array. If $A[\frac{n}{2}] < \frac{n}{2}$, then by the same logic no element in the first half of the array can satisfy the condition.
While the algorithm has not terminated, we discard the half of the array that cannot hold an answer, and recurse on the other half applying the same check on the new array's median. If we are left with an empty array, we return false.

At each step we do a single comparison and discard at least half of the remaining array (or terminate), so this algorithm takes $O(\log n)$ time.

13 My Dog Ate My Homework

One morning, you wake up to realize that your dog ate some of your CS 170 homework paper, which is an $\ell \times w$ rectangular grid of squares. Some of the squares have holes chewed through them, and you cannot use paper that has a hole in it. You would like to cut the paper into pieces so as to separate all the tattered squares from all the clean, un-bitten squares. You want to do this so that you can save as much of your work as possible.

For example, shown below is a 6×4 piece of paper where the bitten squares are marked with *. As shown in the picture, one can separate the bitten parts out in exactly four cuts.



(Each *cut* is either horizontal or vertical, and of one piece of paper at a time.)

Formally, the problem is as follows:

> **Input:** Dimensions of the paper $\ell \times w$ and an array $A[i, j]$ such that $A[i, j] = 1$ if and only if the ij^{th} square has holes bitten into it.
>
> **Goal:** Find the minimum number of cuts needed so that the $A[i, j]$ values of each piece are either all 0 or all 1.

Design a DP-based algorithm to find the smallest number of cuts needed to separate all the bitten parts out in $O(\ell^3 w^3)$ time.

Solution:

(a) **Subproblem Definition:** We define $B[i_1, j_1, i_2, j_2]$ to be the minimum number of cuts needed to separate the sub-matrix $A[i_1 \leq i_2, j_1 \leq j_2]$ into pieces consisting either entirely of bitten pieces or clean pieces.

(b) **Recurrence Relation:**

$$B[i_1, j_1, i_2, j_2] = \min \begin{cases} 0, & \text{if all entries of } A[i_1 \dots i_2, j_1 \dots j_2] \text{ are equal} \\ 1 + B[i_1, j_1, i_1 + k, j_2] + B[i_1 + k + 1, j_1, i_2, j_2] & \text{for any } k \in \{1, \dots, i_2 - i_1\} \\ 1 + B[i_1, j_1, i_2, j_1 + k] + B[i_1, j_1 + k + 1, i_2, j_2] & \text{for any } k \in \{1, \dots, j_2 - j_1\} \end{cases}$$

Alternatively, you could have also encapsulated the 0 base case in all single-square pieces, and determined if a piece was pure via the merging, see below.

(c) **Subproblem Order:** we solve them in increasing order of $(i_2 - i_1 + 1)(j_2 - j_1 + 1)$. In other words, we solve all the smallest subproblems first (e.g., containing one square) and build our DP array up to our result $B[1, \ell, 1, w]$, which covers the entire paper.

(d) **Runtime Analysis:** Two answers are acceptable: $O((\ell + w)\ell^2 w^2)$ and $O(\ell^3 w^3)$

We have $O(\ell^2 w^2)$ total subproblems: $O(\ell w)$ possibilities for (i_1, j_1), and $O(\ell w)$ possibilities for (i_2, j_2). For each subproblem, we examine up to m possible choices for horizontal splits and n possible choices for vertical splits. A single split consideration will result in two smaller subproblems, which we can assume have already been solved, so we just need to find the best split, which takes $O(\ell + w)$ time.

In addition, for a subproblem, we also want to check the base case for if the piece is "pure" (contains only clean paper, or contains only bitten paper). Brute force checking this takes $O(\ell w)$ time, for a total subproblem time of $O(\ell w + (\ell + w)) = O(\ell w)$.

Thus, the overall (accepted) runtime is $O(\ell^2 w^2) \cdot O(\ell w) = O(\ell^3 w^3)$.

However, this $O(\ell w)$ factor per subproblem can be reduced to $O(\ell + w)$ (this is not required to receive full points). We can precompute the purities of every single possible subrectangle and store them in a table. Brute-force performs the pre-computation in $O(\ell^3 w^3)$ time, but using prefix sums allows us to do this in just $O(\ell w)$ time. So to solve our recurrence relation, if we can determine purity/impurity in $O(1)$ time (after doing some pre-computation), then we can reach an overall time of $O((\ell + w)\ell^2 w^2)$.

Alternatively, we can initialize all min-cut values of single square pieces to be 0. Then, if it is possible to have some cut such that both resulting pieces have min-cut values of 0, and both resulting pieces are of the same type (clean-only or bitten-only, and we can take any sample of either and compare them), then we ourselves are a pure piece. This would allow you to avoid the entire pre-computation business as mentioned before, and still achieve a runtime of $O((\ell + w)\ell^2 w^2)$.

(e) **Space Complexity Analysis:** We have to store the entire DP array for our recurrence relation to work, so the space complexity is $O(\ell^2 w^2)$.

