CATTU
EXAM 3

PROOF BY CONTRADICTION: ⟷ **PROOF BY SWAPPING:**

- assume G.C. not optimal
- let opt. soln differ from G.C. @ 1st choice
- show we can construct new opt. soln. matching G.C.
- contradiction!

- consider opt. soln that doesn't make G.C.
- swap opt. choice w/ G.C.
- show that swap doesn't worsen the soln's quality
- convert opt. soln → G.C.

CORRECTNESS:

- by GCP, we assume optimal soln contains the GC.
- after taking GC, remaining problem is smaller but of the same type.
- by strong induction, let's prove this greedy alg is optimal.

v1: Recursion
max{ coins([...]) ; A[i]+coins(S[i:]) }
- many overlapping subp.

v2: Memoization (top down)
- store answers in DAA
- recursive calls don't repeat work

v3: Pure DP
- start from smaller problems
- build in reverse topological order

GREEDY ALG: best option doesn't block optimality

- make 1st decision locally optimally, & @ least 1 global optimal soln will be consistent
- order input in some way then apply GC. repeat.

STEPS:

greedy choice	greedy remaining	choice problem	repeated up

ind. step

- define what you're inducting on
- base case
- IH (assume greedy alg produces opt. soln for ... fewer than n)
- inductive step

1) GREEDY CHOICE: schedule earliest finishing time 1st

2) GCP: ∃ at least 1 optimal soln containing greedy choice

3) PROOF OF GCP: compare some optimal soln that doesn't make GC, prove you can make it @ least as optimal by swapping to make GC.

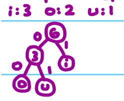
methods of prove [• proof by contradiction • proof by swapping

*preserves feasibility (satisfies constraints) "since ..., it covers all..."

4) PROOF OF ALG BY IND: argue that remaining problem is of some form (self-reduction) and that solving smaller problems optimally & adding 1st choice back in gives optimal soln. can use contradiction within.
→ recursively repeating GC, self reduction

- BASE CASE
- RECURSIVE STEP

5) GIVE ALG & RUNTIME (implement GC)



code:
i: 1
o: 00
u: 01

HUFFMANS ALG: encode letters as binary strings
GREEDY CHOICE: pick smallest frequencies as siblings
- prefix-free (no code word is prefix of others)
- minimize total weighted path length

COMMON GREEDY PATTERNS:

- interval scheduling max # non-overlapping → GC: earliest finish time
- Dijkstras pick closest unvisited node
- coin change use largest coin ≤ remaining amt.
- scheduling w/ deadlines highest profit & place job in latest possible slot

DYNAMIC PROGRAMMING: overlapping subproblems

SUBPROBLEM: memo definition (domain, dimensions) M[i] = ...
M[i] = ... # dims depends on states (weight, len, ind)

RELATIONSHIP: DP transition (recursion equation)
M[i] = max/min{

TOPOLOGICAL SORT: argue graph formed in R is acyclic, & table can be filled iteratively in reverse topo. order

BASE CASES: initialize

OUTPUT: which entry of M to output
M[...]

TIME: runtime, often product of (# entries in M) × (work per entry)
O(...)

DP SP:

DAG SSSP:
S: x(v) = dist. to v
R: x(v) = min{x(u) + w(u,v)} (incoming edges)
T: rev. topo order of G
B: x(s) = 0
O: whole table
T: O(|V|+|E|)

BF DP: handles cycles w/ extra dim.
S: x(v,k): weight of SP from s to v using ≤k edges
R: x(v,k) = min{ min{x(u,k−1) + w(u,v)} ← use kth path ; x(v,k−1) ← don't use kth path }
B: x(s,0) = 0 x(v,0) = ∞ for v ≠ s

APSP (FW): arbitrary ordering.
S: d(u,v,k) = SP using only vertices in {u,v} ∪ {1,...,k}
R: d(u,v,k) = min{ d(u,v,k−1) ← no vertex k in interm. ; d(u,k,k−1) + d(k,v,k−1) ← yes vertex k }
T: O(|V|³)·O(1)

COMMON STATES:
- post resource
- L+R endpt
- vertex + # edges
] 2

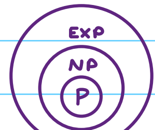
COMMON DP PATTERNS:

- knapsack dp[i][j] = best among 1st i items → NP-complete (open q)
- weighted intervals DP on jobs sorted by finish times
- edit distance [i][j] = cost to convert 1st i chars to 1st j chars
- LCS grid [i][j] = LCS of prefixes :i & :j
- LIS [i] = longest incr. ending @ i
- grid path from top-left or bottom right. DP cell = best from neighbors
- DAG SP/LP topo/rev. topo order. recurrence on neighbors
- trees combine child subtrees
- coin change min #coins / count ways to make amount
- sandwich cutting T(ℓ) max val cutting sandwich of len ℓ
- subset sum decision problem. T(ℓ,i) = T if subset $a_i \ldots a_n$ sum to ℓ. not poly time - input size isn't L, it's logL (bits). if L is huge (2ⁿ), O(nL) is actually exponential (NP complete), pseudopoly
- s-t reachability in NP & not known to be NP-complete

COMPLEXITY CLASS: collection of decision problems w/ some property

① P: problems solveable in O(nᵏ) for constant k where n in input space

② NP: problems verifiable in O(nᵏ) for some constant k.
- VERIFIER V(x,c): poly-time alg that takes instance x & CERTIFICATE c then returns Y/N



NP-HARD: B is NP-hard if for all problems A ∈ NP, A ≤p B
NP-COMPLETE: NP-hard & in NP
- if A ≤p B & B is in P/NP/EXPTIME, then so is A
- if A ≤p B & A is in NP-hard, then so is B.

REDUCTIONS:
- convert one problem into another
- A reduce to B → B @ least as hard as A
- A ≤p B if can solve B in poly, can solve A
- used to compare difficulty of problems
- are transitive!

Ex: Partition ≤ Subset Sum

A ≤p B
B = upper bound for A → A can't be harder than B (≤)
A = lower bound for B → B can't be easier than A (≥)

- B∈P → A∈P
- A∉P → B∉P

- if A is easy, B might still be hard
- if B is hard, A might still be easy
- if B runs in linear, A may still be poly

NP PROOF:
- certificate
- verifier: returns Y/N based on input & certificate
- argue poly R.T.

POLYNOMIAL TIME
- running time = # of bits of input size
Ex: array of n #, each up to 1 mil. poly depends on n

NP-COMPLETE EXS:
- clique
- partition
- 0-1 knapsack
- 3-colorability
- 3SAT
- subset sum

NP-hard PROOF:
- "we reduce from ___, which is NP-complete"
- common choices: 3-SAT, 3-COLOR, CLIQUE
- construct reduction
- correctness argument → show YES maps to YES, NO maps to NO (both directions)
- poly time RT argument

PSEUDOPOLY TIME
- polynomial in numeric vals of inputs
- time ≤ O(1) deg. poly in input size & int inputs
Ex: array of n #, each up to 1 mil. pseudo poly depends on time prop. to 1 mil. which can be exponential.

(k) [4 points] If P = NP, then which of the following must be true?
Select **all** correct answers.

- ■ There is an NP-complete problem that can be solved in polynomial time.
- ■ Every NP-complete problem can be solved in polynomial time.
- ■ There is an NP-hard problem that can be solved in polynomial time.
- ☐ Every NP-hard problem can be solved in polynomial time.

HALTING PROBLEM:
in: (P,x) alg, input
out: yes if P halts on x
no otherwise
→ NP hard & undecideable (not in NP)

UNDECIDEABLE: if no alg you can makes solves the problem.
thm: Halt is undecideable (prove w/ Diagonalization)
(no alg. can solve in poly)

TOTALITY:
in: P
out: yes if P halts on all inputs
no o.w.
thm: totality is undecideable
Pf: Halt ≤p Totality
assume alg T decides totality
construct H, a decider for Halt
H(P,x):
1) define Q(y) := ignore input. run P on x. output yes (as long as no ∞ loop)
2) return T(Q) (whether P halts)

EQUIV:
in: P, Q
out: yes if P, Q output yes on same inputs
no o.w.
thm: equiv is undecideable
Pf: Halt ≤p Equiv
assume alg. E decides Equiv
H(P,x):
1) define Q(y): "ignore input. run P(x), output yes"
2) define R(y): "ignore input, output yes" always
3) return E(Q,R)

4. Consider two decision problems A and B. The problem $A \cup B$ asks whether its input is a YES instance of A *or* a YES instance of B. Similarly, the problem $A \cap B$ asks whether its input is a YES instance of A *and* a YES instance of B.

Circle all necessarily true statements. NP & P

(a) If $A, B \in$ P, then $A \cup B \in$ P. YES if poly time solveable, can just solve both
(b) If $A, B \in$ P, then $A \cap B \in$ P. YES solve both
(c) If $A, B \in$ NP, then $A \cup B \in$ NP. YES: certificate tells us which problem to use & has certificate for that problem.
(d) If $A, B \in$ NP, then $A \cap B \in$ NP. YES

Recall that in PARTITION, we are given a list of numbers A and asked whether it can be partitioned into two lists with the same sum. This problem is NP-complete.

We can define a decision problem 0-1 KNAPSACK as follows: we are given a capacity S, a target value V, and a list of items, which each have a *size* s_i and a *value* v_i. We are asked whether there's a subset of items with total size at most S and total value at least V. (The "0-1" in the name comes from the fact that each item can be taken zero or one times, but not multiple times or a fractional value.)

Prove that 0-1 KNAPSACK is NP-hard by describing a reduction from PARTITION to 0-1 KNAPSACK. NP-HARD: KNAPSACK partition ≤p knapsack

Solution: Given an instance A of PARTITION, we construct an instance of 0-1 KNAPSACK. For each element a_i of A, we create an item with $s_i = v_i = a_i$. Let $T = \sum a_i$ be the sum of all of the numbers. Our capacity and target value are $S = V = T/2$.

If we started with a YES instance of PARTITION, then there is a solution to the instance of 0-1 KNAPSACK: take the items corresponding to either set in a valid partition, which have total value and total size $T/2$.

Conversely, if this is a YES instance of 0-1 KNAPSACK, then we can find a partition: one side is the elements of A corresponding to items taken in the solution to 0-1 KNAPSACK, and the other side is everything else. Both of these sets have sum $T/2$.

Incidentally, 0-1 KNAPSACK is in NP, so it is also NP-Complete.

Partition

- Input: Sequence of n positive integers $A = \{a_1, \ldots, a_n\}$.
- Output: Is there a bipartition of A into two subsets with equal sum?
- To solve Partition, we can **reduce** it to Subset Sum. We efficiently convert instances of Partition into instances of Subset Sum, and then we can use any algorithm for Subset Sum to solve them.

Subset Sum NP Complete

- Input: Sequence of n positive integers $A = \{a_1, \ldots, a_n\}$, another integer L.
- Output: Is there a subset of A that sums exactly to L? (i.e., $\exists A' \subseteq A$ s.t. $\sum_{a \in A'} a = L$?)
- This is a **decision problem**. Answer is YES or NO, TRUE or FALSE
- Example: $A = \{2, 5, 7, 8, 9\}$, $L = 21$ is YES ($L = 5 + 7 + 9$ is a valid solution, or **witness**)

(a) Prove that ACCEPT is *recognizable*. That is, prove that there exists an algorithm $\mathcal{B}$ such that $\mathcal{B}(\mathcal{A}, x) =$ YES iff ACCEPT$(\mathcal{A}, x) =$ YES. RECOGNIZEABLE

Solution: $\mathcal{B}(\mathcal{A}, x)$ runs $\mathcal{A}(x)$ and outputs the result (if any). Then $\mathcal{B}(\mathcal{A}, x) =$ YES iff $\mathcal{A}(x) =$ YES iff ACCEPT$(\mathcal{A}, x) =$ YES as desired.

(b) Prove that ACCEPT is undecidable by reducing from HALT.

Solution: UNDECIDEABLE

Input is $(\mathcal{A}, x)$

Create a new algorithm $\mathcal{A}'$, which on input y, runs $\mathcal{A}(y)$ and then outputs YES.

Output is $(\mathcal{A}', x)$

Now $\mathcal{A}'(x) =$ YES iff $\mathcal{A}(x)$ halts, i.e. ACCEPT$(\mathcal{A}', x) =$ HALT$(\mathcal{A}, x)$. HALT is undecidable, and HALT $\leq_p$ ACCEPT, so ACCEPT is undecidable also.

LONGEST INCREMENTING SUBSEQUENCE:

LIS DP

[8, 5, 10, 6, 7, 9] → [5, 6, 7, 9]

S: T(i, j) = len of LIS of A[i : n] w/ ($0 \le i \le n$, $-1 \le j \le i-1$) numbers larger than A[j]

R: T(i, j) = max { 1 + T(i+1, i) if j = -1 or A[i] > A[j] ; T(i+1, j) }

:

O: T(0, -1)

Alternate:

S: T(i) = len of LIS of A[i : n] using A[i]

R: T(i) = 1 + max { T(j), A[i] < A[j] } , i < j < n

SRTBOT for the coin row problem

COIN ROW

- **Input:** an array A denoting the values of the coins.

S Let $M(i)$ be the maximum value of coins we can pick from $A[i:]$, for $i \in \{0, \ldots, n-1\}$.

R $M(i) = \max\{M(i+1), A[i] + M(i+2)\}$, for $i = \{0, \ldots, n-1\}$.

T Each $M(i)$ depends on $M(i')$ with larger index $i' > i$. Therefore, we can compute $M(i)$ in decreasing order of i.

B $M(i) = 0$ if $i \ge n$ (i.e. if $A[i:]$ is empty).

O Output is $M(0)$. Note, however, that we may want the actual list of coins, which we will discuss next.

T There are n values $M(i)$. Each takes $O(1)$ to compute for a total of $O(n)$.

S $T(k, u, v)$ = minimum weight of a path from u to v through vertices in $[k+1] \cup \{u, v\}$. The notation $[k+1]$ means 0, 1, …, k.

APSP: FLOYD WARSHALL

R We have the choice of using node k or not. A shortest path that uses k as an intermediary node is formed of a shortest path from u to k using only $[k]$, i.e., 0, 1, …, $k-1$, as possible intermediary nodes plus a shortest path from k to v also restricted to $[k]$ as possible intermediary nodes. Therefore,

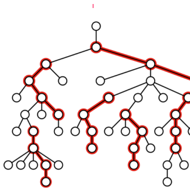
$$T(k, u, v) = \min \begin{cases} T(k-1, u, k) + T(k-1, k, v) \\ T(k-1, u, v) \end{cases}$$

T Decreasing k.

B $T(-1) = w$ (i.e. $T(-1, u, v) = w(u, v)$ for all u and v). There are no intermediate vertices between u and v.

O $T(|V|, \cdot, \cdot)$ contains all pairs of shortest paths without any restrictions.

T The table T has size $(|V|+1) \times |V| \times |V| \in O(|V|^3)$. Each entry takes $O(1)$ work for a total of $O(|V|^3)$.



GREEDY

(a) Describe, analyze, and prove correctness of an $O(n)$-time greedy algorithm to compute the maximum number of disjoint paths that can fit in the tree. Your algorithm is given a rooted tree T and an integer k as input, and it should output the largest possible number of disjoint k-edge paths directed towards the root in T. Do not assume that T is a binary tree. For example, given the tree above as input, your algorithm should return 8. Note that you only need to return the number of paths and not the location of the paths.

Prove the greedy choice property you rely on, and argue the correctness and runtime of your algorithm.

a) GC: consider node x w/ height k.

use a path starting from leaf & ending @ x.

GCP: there ∃ @ least 1 optimal soln containing GC.

Pf of GCP: consider optimal soln S that doesn't make GC.

if S doesn't use x at all, we can add new path

if S does use x, can shift path down

Alg:

· recursively compute height of each vertex

· when node x has height k,

1. add 1 to counter

2. remove x & its subtree from tree

O(n) post order traversal.

Pf by induction:

BC: height < k

Ind Step: consider opt. soln

by GCP, can assume S makes same 1st step as G.

say we removed node x to get T.

we can consider our alg. to have run on T' up to this point

by IH, |G'| ≥ |S'| → |G| = |S|

greedy does as good as optimal

Proof of correctness:
We prove by induction on the number of intervals.
Base case: If no intervals remain, selecting no intervals is optimal.
Inductive hypothesis: Assume the greedy algorithm is optimal for all instances with fewer than n intervals.
Inductive step: Consider an instance with nnn intervals. By the greedy choice property, there exists an optimal solution that includes the greedy choice. After selecting it, the remaining intervals form a smaller instance of the same problem. By the inductive hypothesis, the greedy algorithm is optimal on the remaining instance, so the full solution is optimal.

GREEDY: CORRECTNESS

(b) [15 points] Describe modifications to your algorithm in Part (a) so that it runs in $O(n \log k)$ time. Analyze its runtime, but you need not prove correctness. You need not re-state your SRTBOT other than steps that you modify (if applicable). Correct $o(nk)$ algorithms will be eligible for partial credit.

Solution: Store $X(j)$ in a max-heap cross-linked with a DAA. At each iteration of the R step, we access the max m of the heap and compute $X(i) = \lfloor m/2 \rfloor + p_i$. We then insert $X(i)$ into index i in the DAA, insert it into the heap, and remove $X(i-k)$ from the heap (if it exists).

OR MAX HEAP/AVL

Store $X(j)$ in an AVL sequence augmented with the max in each subtree. At each iteration of the R step, we access the root augmentation m and compute $X(i) = \lfloor m/2 \rfloor + p_i$. We then append the new $X(i)$ and, if there are more than k elements, delete the first.

For Y, we instead use a min-heap or min augmentation, compute $Y(i) = 2m$, and store $Y(i) - p_i$.

Let L be a decision language. Prove that L is decidable iff both L and its complement L are recognizable.

DECIDEABLE

Solution: The forward direction is easy; a decider for L is also a recognizer for L, and flipping its output produces a decider (and hence recognizer) for $\overline{L}$.

Conversely, suppose L and its complement are both recognizable, say by algorithms $\mathcal{A}$ and $\mathcal{B}$ respectively. We decide L as follows:

- Input is x
- Run $\mathcal{A}(x)$ and $\mathcal{B}(x)$ in parallel
- If $\mathcal{A}(x)$ says YES, output YES
- If $\mathcal{B}(x)$ says YES, output NO

*can't loop forever b/c together they cover all inputs

will eventually say YES from one of them.

L and $\overline{L}$ partition all inputs, so one of $\mathcal{A}(x)$ or $\mathcal{B}(x)$ outputs YES. The above algorithm is therefore a decider. It outputs YES iff $\mathcal{A}$ does, so it decides L.

Each city has a *disapproval score*, which is an integer. The disapproval scores **may be zero, negative, or positive**. The disapproval score for the j^{th} city on row i is $A[i][j]$. For example, if $n = 4$, the disapproval scores may be as shown above.

Over all possible journeys, Srini wants to know the **maximum product** of disapproval scores of the cities he visits. In our example above for the particular values of A, the maximum disapproval score corresponds to the path marked and has value $(-1)(-2)(-2)(-1) = 4$.

Design an $O(n^2)$-time Dynamic Programming algorithm that returns the cities on the path with maximum product of disapproval scores. You may assume that all arithmetic operations take constant time.

Solution: DP: Tmax & Tmin

As with the arithmetic parenthesization problem in R19, we don't know when we want to maximize the product of a subpath and when we want to minimize it. So we'll do both.

Subproblems: For $0 \le j \le i \le n$, let $T_{max}(i, j)$ and $T_{min}(i, j)$ be the maximum and minimum (respectively) possible products of disapproval ratings of southward journeys starting at $C[i][j]$.

Note that we allow i to be n, which is a fake city south of all the real cities—this makes our base case simpler, since we can consider the 'journey' that starts at such a fake city and does nothing to have product 1.

Relate: There are two places we could go from $C[i][j]$. Whether we need to maximize or minimize the product after leaving $C[i][j]$ depends on the sign of $A[i][j]$. We get the following recurrences.

$$T_{max}(i, j) = \begin{cases} A[i][j] \cdot \max(T_{max}(i+1, j), T_{max}(i+1, j+1)) & \text{if } A[i][j] > 0 \\ 0 & \text{if } A[i][j] = 0 \\ A[i][j] \cdot \min(T_{min}(i+1, j), T_{min}(i+1, j+1)) & \text{if } A[i][j] < 0 \end{cases}$$

$$T_{min}(i, j) = \begin{cases} A[i][j] \cdot \min(T_{min}(i+1, j), T_{min}(i+1, j+1)) & \text{if } A[i][j] > 0 \\ 0 & \text{if } A[i][j] = 0 \\ A[i][j] \cdot \max(T_{max}(i+1, j), T_{max}(i+1, j+1)) & \text{if } A[i][j] < 0 \end{cases}$$

It would also work to take the max or min of all four possible products of $A[i][j]$ and a relevant subproblem.

Topological order: $T_{max}(i, j)$ and $T_{min}(i, j)$ only depend on subproblems $T_{max}(i', j')$ and $T_{min}(i', j')$ with $i' > i$.

Base case: Our recurrence breaks down in the last row (of fake cities) when $i = n$. There we have $T_{max}(n, j) = T_{max}(n, j) = 1$ for all j.

Original problem: The maximum possible product is $T_{max}(0, 0)$. We can follow use parent pointers to construct the path, by recording which option was the best for each subproblem.

Time: It takes $O(1)$ time to solve each of $O(n^2)$ subproblems, so the runtime is $O(n^2)$.

Problem 1. [20 points] **DP** (1 part)

iven a string S of length n where each character is one of the 26 English characters, design an $O(n)$ ynamic programming algorithm to compute the length of the longest strictly-increasing sub-sequence in S ı alphabetical order).

ake sure to use the SRTBOT framework.

DP: state = constant

olution: Treat the characters as integers (1-26).

S Let $T(i, k)$ be the length of a LIS in $S[i:]$ with start value $\ge k$. $(0 \le i \le n, 1 \le k \le 27)$

R $T(i, k) = \max \begin{cases} T(i+1, k) \\ 1 + T(i+1, S[i] + 1) & \text{if } S[i] \ge k \end{cases}$

T Decreasing i.

B $T(n, k) = T(i, 27) = 0$ for all i, k.

O $T(0, 1)$.

T The table T has size $27(n+1) = O(n)$. Each entry takes $O(1)$ to compute for a total of $O(n)$.

(d) [10 points] Give a full greedy algorithm to solve the problem. Prove correctness and analyze runtime.

Solution: Algorithm $\mathcal{A}$:

GREEDY: alg proof w/ ind/contra.

1. Order the intervals by b_i. — $O(n \log n)$
2. Set $S \leftarrow \varnothing$ and $b \leftarrow -\infty$. — $O(1)$
3. Scan the intervals in order: When processing $[a_i, b_i]$, 'add b_i to S and set $b \leftarrow b_i$' if $a_i > b$. — $O(n)$
4. Output S. — $O(1)$

Total runtime: $O(n \log n)$. To prove correctness, we use strong induction on n.

1. Let GREEDY be the output by $\mathcal{A}$, and S be an optimal set of points that contains the greedy choice b_k (which exists by (b)).
2. For contradiction, assume that $|\text{GREEDY}| > |S|$.
3. Let $\mathcal{I}'$ be the set of intervals not covered by b_k, i.e. intervals that start after b_k.
4. Note that after inserting b_k into S, $\mathcal{A}$ ignores all intervals covered by b_k, and the repeats the process on $\mathcal{I}'$, solely for which $\mathcal{A}$ would output GREEDY $\setminus \{b_k\}$.
5. By the induction hypothesis, $\mathcal{A}$ is optimal for $\mathcal{I}'$ (as $|\mathcal{I}'| < n$) and hence

$$|\text{GREEDY} \setminus \{b_k\}| \le |S \setminus \{b_k\}|.$$

6. Therefore, $|\text{GREEDY}| \le |S|$, a contradiction.

Problem 5. [45 points] **Reduction** (3 parts)

Erik, Mohsen, and Brynmor want to celebrate the end of the semester by throwing a party. However, among their mutual friends, there are certain pairs of friends that cause trouble together.

The Professors still want to celebrate, so they come up with a plan: host 3 parties (i.e. Erik, Mohsen, and Brynmor each throw their own party, so the troublemakers can go to separate parties and everyone can still celebrate).

REDUCTION

We define INVITE-FRIENDS as the decision problem: given a list of n friends and a list of m pairs of friends who cause trouble together, is it possible to invite everyone to one of the three parties.

For example, if the Professors are friends with [Maggie, Selina, Daph, Ragulan] and the trouble making pairs are (Maggie, Selina), (Selina, Daph), (Daph, Ragulan), (Selina, Ragulan), one solution to the problem is: Maggie and Ragulan could go to Erik's party, Daph could go to Mohsen's party, and Selina could go to Brynmor's party.

Unfortunately, INVITE-FRIENDS is NP-complete. Poor Professors! To console them, you will write a great proof arguing that the INVITE-FRIENDS is NP-complete.

(a) [20 points] Show that INVITE-FRIENDS $\in$ NP.

Solution: To show that INVITE-FRIENDS $\in$ NP, we show that we can verify a certificate in polynomial time.

- Let the certificate be three lists of friends: one list of attendees for each party.
- Verifier: For each party and all pairs of friends invited to that party, check if any pair will cause trouble. If any pair of friends at the same party cause trouble, return NO. Otherwise, return YES.
- If the professors have n friends, the certificate will be of size $O(n)$.
- Runtime for the verifier algorithm: For each of the $O(n^2)$ pairs of friends at the same party, scan the $O(m)$ list of friends that cause trouble. Thus, the verifer takes $O(n^2 m)$, which is polynomial in the size of the instance of the problem (which is $\Theta(n + m)$).

(b) [20 points] Show that INVITE-FRIENDS is NP-hard by reducing from 3-COLOR , an NP-complete problem.

3-COLOR is the following decision problem: Given a graph G, is it possible to color the vertices of G using 3 colors, such that neighboring vertices are not colored the same color?

3 col ≤p invite friends

Solution: In order to show INVITE-FRIENDS is NP hard, we must transform an arbitrary instance of 3-COLOR into an instance of INVITE-FRIENDS .

Given graph $G = (V, E)$, we can let each vertex, v, be a friend in INVITE-FRIENDS . Each edge (u, v) can represent a pair of friends that cause trouble in INVITE-FRIENDS . So a valid coloring in 3-COLOR becomes a valid party assigment in INVITE-FRIENDS .

Thus, we have transformed the instance of 3-COLOR into an instance of INVITE-FRIENDS .

It takes $O(|V| + |E|)$ to iterate through the vertices and edges, so this reduction takes polynomial time.

Thus, because 3-COLOR is NP-complete, and we can reduce 3-COLOR to INVITE-FRIENDS in polynomial time, INVITE-FRIENDS is NP-hard.

(a) Given strings A, B and C, design a DP algorithm that checks in $O(|A||B|)$ time whether there is a way to merge A and B to produce C. You may assume that $|C| = |A| + |B|$.

PSETS

For example, if $A = abab$, $B = aabb$ and $C = abaaabbb$ then the output should be TRUE as $C = aba + aabb + b$ where the first and last substrings come from A and the middle one comes from B.

Make sure to use the SRTBOT framework.

DP: merge strings

Solution:

We assume that $|A| = n, |B| = m$ and $|C| = n + m$.

1. **Subproblems** Let $T(i, j)$ = TRUE *iff.* one can merge $A[i:]$ and $B[j:]$ into $C[i+j:]$. Index i runs from $0, \ldots, n$ and index j runs from $0, \ldots, m$.
2. **Relate** We consider the case where the next letter will come from A and B by considering incrementing either the pointer at i or the pointer at j, respectively.
$$T(i, j) = \vee \begin{cases} T(i+1, j) \text{ if } C[i+j] == A[i] \\ T(i, j+1) \text{ if } C[i+j] == B[j] \end{cases}$$
We also handle edge cases–if one of our conditional statements requries checking an index out of range, we assume it to be FALSE.
3. **Topo. Order** by $i + j$
4. **Base** $T(n, m)$ = TRUE.
5. **Original** $T(0, 0)$.
6. **Time** The table T has size $(n+1)(m+1) \in O(nm)$. each entry takes constant time to compute, for a total of $O(nm)$.

(a) Frieren has a row of treasure chests numbered 1 through n. There is an unknown s such that chests 1 through s are *safe*, and the rest are *mimics*. Frieren can open chest i and learn whether or not it is safe by spending a positive amount m_i of mana.

Frieren must compute s without opening more than k mimics, given k and all of the m_i as inputs. Frieren will use the most efficient algorithm, in terms of the worst-case total mana cost. For example, if $k = 1$, then the only correct algorithm is to open all chests in increasing order of i, which costs $\sum m_i$.

Design a DP that computes the worst-case mana cost of Frieren's algorithm. Your DP should run in time $O(n^3 k)$. Use the SRTBOT framework, but you need not prove correctness beyond that.

Hint: Discuss with friends or ask course staff to make sure you have the right subproblem definition!

Solution: Give as hint: $M(a, b, c)$ is the worst-case cost of Frieren's algorithm on input $(c, m[a:b])$

S $M(a, b, c)$ is the worst-case cost of Frieren's algorithm on input $(c, m[a:b])$, where $0 \le c \le k$ and $0 \le a \le b \le n$

R $M(a, b, c) = \min_{a \le q < b} (m_q + \max\{M(a, q, c-1), M(q+1, b, c)\})$

T Decreasing $b - a$

B $M(a, b, 0) = \infty$ if $b > a$; $M(a, a, c) = 0$

DP: interval search

O $M(0, n, k)$

T There are $\binom{n+1}{2} \times (k+1)$ subproblems, each of which takes $O(n)$ time to compute, for a total of $O(n^3 k)$ time.

(b) Modify your DP so that it runs in $O(n^2 k \log n)$ time. (Warning: this is quite hard.)

Solution: Give as hint: Notice that M is a decreasing function of a and an increasing function of b. What does this say about how to leverage **binary search**?

Per the hint, we observe that the difference between the terms in the max is monotone. This allows us to remove the max entirely by simply conditioning on whether q is small or large. We binary search for 0 in $M(q+1, b, c) - M(a, q, c-1)$. This gives a q^* for which

$$M(a, b, c) = \min \begin{cases} \min_{a \le q < q^*} (m_q + M(q+1, b, c)) \\ \min_{q^* \le q < b} (m_q + M(a, q, c-1)) \end{cases}$$

We store M redundantly in $O(nk)$ AVL BSTs. $T(b, c)$ maps a to $m_{a-1} + M(a, b, c)$, where $m_{-1} = 0$ for sake of notation. $T'(a, c)$ maps b to $m_b + M(a, b, c)$. We augment each $T(b, c)$ and each $T'(a, c)$ with the min function. Now the top term is a prefix

(a) Let $A = [a_0, \ldots, a_{n-1}]$ be a sequence of n positive integers with sum $m = \sum A$. Say that a partition $\{A_1, A_2, A_3, A_4\}$ of A into four subsequences has **quality** $Q = \min\{\sum A_i \mid i \in \{1, 2, 3, 4\}\}$. Describe an $O(m^3 n)$-time algorithm to find the maximum possible quality of a partition of A into four subsets.

Note: *After* solving this part with a SRTBOT in $O(m^3 n)$ time, think about how you can implement it with an algorithm whose runtime is *also* bounded by $O(4^n)$. It is not necessary for this part and should not change your DP or SRTBOT fundamentally, but your code in Part (b) will need to meet both runtime upper bounds.

Solution: DP: partition 4 subsets

S $x(k, s_1, s_2, s_3)$ (where $0 \le k \le n$ and $0 \le s_i \le m$) is True if it is possible to partition $A[k:]$ into four subsets A_1, A_2, A_3, A_4 with $s_j = \sum A_j$ for all $j \in \{1, 2, 3\}$, and false otherwise

R Integer a_k must be placed in some partition. Guess!

$$x(k, s_1, s_2, s_3) = \text{OR} \begin{cases} x(k+1, s_1 - a_k, s_2, s_3), & \text{if } a_k \le s_1 \quad (\text{add } a_k \text{ to } A_1) \\ x(k+1, s_1, s_2 - a_k, s_3), & \text{if } a_k \le s_2 \quad (\text{add } a_k \text{ to } A_2) \\ x(k+1, s_1, s_2, s_3 - a_k), & \text{if } a_k \le s_3 \quad (\text{add } a_k \text{ to } A_3) \\ x(k+1, s_1, s_2, s_3), & \text{always} \quad (\text{add } a_k \text{ to } A_4) \end{cases}$$

T Decreasing k ($x(k, s_1, s_2, s_3)$ only depends on subproblems with strictly larger k)

B $x(n, 0, 0, 0)$ = True (can partition zero integers into zero sum subsets)
$x(n, s_1, s_2, s_3)$ = False for any $s_1, s_2, s_3 > 0$ (cannot partition zero integers into any subset with positive sum)

O Let $m(0, s_1, s_2, s_3) = \begin{cases} \min\{s_1, s_2, s_3, m - s_1 - s_2 - s_3\} & \text{if } x(0, s_1, s_2, s_3) \\ -\infty & \text{otherwise} \end{cases}$

Solution to original problem is then given by $\max\{m(0, s_1, s_2, s_3) \mid 0 \le s_i \le m\}$
(Can also store parent pointers to reconstruct each subset)

T There are $O(m^3 n)$ subproblems. Each takes $O(1)$ time to compute. O step takes $O(m^3)$ time. Total is $O(m^3 n)$, which is **pseudopolynomial** in the size of the input.
If solve order starts with $x(n, 0, 0, 0)$ and uses the reversed dependency graph, the False subproblems can be ignored. There are only $O(4^n)$ True subproblems, so the runtime can WLOG be bounded by $O(4^n)$.

Greedy Choice Property:

At the earliest start time s∗s^*s∗, selecting the interval that ends latest is safe. Any solution must include an interval starting at s∗s^*s∗, and replacing it with the latest-ending one does not reduce feasibility.

GREEDY: intervals

Algorithm:

Sort intervals by start time. Repeatedly select, among intervals starting at the earliest uncovered time, the one with the largest end, then advance the uncovered time to that end.

Correctness:

By the greedy choice property, there exists an optimal solution containing the greedy choice. Removing covered intervals yields a smaller instance of the same problem. By induction, the greedy algorithm is optimal.

Runtime:

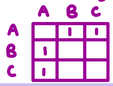
Sorting takes O(nlogn)O(n \log n)O(nlogn); the scan is linear.

CATTU TEST 2

SIMPLE PATH:
- no edge or node duplicates (no loops)

SIMPLE GRAPH:
- no duplicate edges and
- no self loops

{A: [B,C],
B: [A],
C: [A]}



GRAPH: $G=(V,E)$ is set of pairs of vertices V & edges (pairs of vertices) $E \subseteq V \times V \to |E| = O(|V|^2)$

$adj^+(u)$ = set of outgoing edges from u

$adj^-(u)$ = set of incoming edges

$deg^+(u) = |adj^+(u)|$ $deg^-(u) = |adj^-(u)|$

$\sum_{u \in V} deg^+(u) = |E|$

ADJACENCY LIST: $\Theta(|V|+|E|)$ - DAA, hash, LL

ADJACENCY MATRIX: $\Theta(|V|^2)$

PATH: sequence of vertices connected by edges

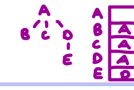
$\delta(u,v)$: shortest path from $u \to v$ (∞ if no path)

single_pair_reachability(G,s,t): True if path $s \to t$

single_pair_shortest_path(G,s,t): SP & distance $\delta(s,t)$

single_source_shortest_path(G,s): SP tree & dist. for all V starting @ s.

SHORTEST PATH TREE:
- contains one path from s to every vertex v reachable from s.
- parent array (predec.)
- can convert b/t SPT & $\delta(s,-)$ in linear time.



REDUCTION: alg. for transferring one problem to another.
- TURING: $A \le_T B$ iff A is solved using B as atomic subroutine.
  - poly time upper bound: cook red.

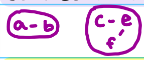
MANY-ONE: $A \le_m B$ iff ∃ some func. f converts input of A to input of B
- if f poly time: KARP so $A \le_p B$ means that f[inputs to A] ↦ [inputs to B]

*if A reduces to B, CANT assume the converse

$A \le_T B$ in time f(n) & B solved in g(n), then A can be solved in $f(n) \cdot g(n+f(n))$

UNDIRECTED GRAPHS:

CONNECTED COMPONENTS: u-v path b/t vertices in same CC



BFS for each node & find CCs → $O(|V|+|E|)$ (touch each node once)

EQUAL WEIGHT GRAPHS:
*for directed & undirected

BFS: $O(|V|+|E|)$
- go in layers
- solves SSSP
- return shortest distances from start node to each reachable node (shortest, in layers)

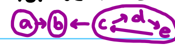
DFS: $O(|E|)$
- not SP
- solves SPR
- reachability
- retrieves set of reachable nodes from start node

FULL DFS: run DFS from every unexplored vertex in G until all are explored. $O(|V|+|E|)$
- **FINISHING ORDER:** mark in order DFS run has explored V & all its neighbors. parent FO is always > than children (child finishes first)

CYCLE DETECTION (full dfs):
- run full DFS. check active vs. finished for nodes
- if an edge goes back to a visited but unfinished node, that node is ancestor in DFS & forms cycle.

DIRECTED GRAPHS:

STRONGLY CONNECTED COMPS (SCCS): ∃ u-v & v-u path for vertices in directed graph



CONDENSATION GRAPH: V = SCCs of graph

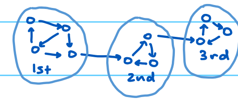
E = ∃ edge from vertex in C_i to vertex in G.

src sink src
a → b ← c.

*CG must be DAGS! use Kosaraju-Shavir to make SCCs

KOSARJV-SHAVIR: finds SCCs $O(|V|+|E|)$
- run full DFS on G^{rev} & record finishing times f[v] for each v in reverse f[v] order.
- for each v in rev. finishing order:
  - set current = v
  - if v = unvisited, run DFS starting from v. for any vertex u w/o leader, set leader[u] = v (grouping SCCS)

intuition: heirarchy of SCCs



← starts from 3rd, then 2, 1.

TOPOLOGICAL SORT: order of nodes s.t. (u,v) ∈ E, then u comes before v in order.
- full DFS, topo order = reverse finishing time.

*must be acyclic, directed graphs

T.O. = a, c, e
a → c → e

WEIGHTED GRAPHS:

GRAPH DUPLICATION: EX: only want to return even # paths
- can use BFS to track even/odd

DUMMY NODES: insert nodes to match weight
*must be positive weight & finite!

O →(3) O →(1) O becomes O → O → O → O → O

CHANGEABLE PQS: $O(|V|\log|V|+|E|)$
- build → generating RT for Dijkstra
- insert
- delete min $O(\log n)$
- decrease key(i, new_key) $O(1)$ ← Fib. heap
  - change val @ i & heapify_up
- store every v∈V keyed by distance estimate
- perform one delete min for every node and one decrease key for every edge

interface:

PRIORITY QUEUE:

interface	HEAP:	AVL:
build(A)	O(n)	O(nlogn)
insert(x)	O(logn)	O(logn)
delete_max() / delete_min()	O(logn)	O(logn)

(implementation: HEAP, AVL)

DIJKSTRA: $O(|V|\log|V|+|E|)$ SSSP on + weight graph
- distance estimates $d[v] = \infty$ except source $d[s] = 0$
- consider nodes in increasing distance
  - for all outgoing edges, relax them.

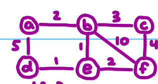
RELAX: if $d[v] > d[u] + w(u,v)$, set $d[v] = d[u] + w(u,v)$

*want to relax each edge exactly once

*doesn't work w/ negative edge weights

δ = pred.
d = actual

- @ end, $d(v) = \delta(v)$ ← path relaxation lemma



d[v]

a	0	0	0	0	0	0	0
b	∞	2	2	2	2	2	2
c	∞	∞	5	5	5	5	5
d	∞	5	5	4	4	4	4
e	∞	∞	3	3	3	3	3
f	∞	∞	12	5	5	5	5

DIJKSTRA DIFFERENT METRICS:

min, +: $[0, \infty)$ → minimizes sum of edge weights along path. SSSP

min, ×: $[1, \infty)$ → minimizes product of edge weights along path

min, max: $(-\infty, \infty)$ → minimizes largest edge along path

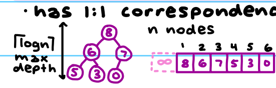
max, min: $(-\infty, \infty)$ → maximizes smallest edge along path

max, ×: $[0, 1]$ → maximizes product of edge weights (all < 1)

TRIANGLE INEQ: for paths: $\delta(a,c) \le \delta(a,b) + \delta(b,c)$

for edge: $w(a,c) > w(a,b) + w(b,c)$

COMPACT BINARY TREE: all nodes as far left as possible
- has 1:1 correspondence w/ array implementation



parent x's children:
- LEFT(x) = 2x
- RIGHT(x) = 2x+1
- PARENT(x) = $\lfloor \frac{x}{2} \rfloor$

HEAPS: compact binary tree implementation of PQ I.P. ✓

MAX HEAP PROPERTY: for x∈Tree, x ≥ left(x), x ≥ right(x)
→ MIN HEAP: opposite (parent = min elt.)

OPERATIONS: find: O(n) → not sorted
- heapify_up(x): O(logn) → swap x up tree while bigger than parent
- heapify_down(x): O(logn) → swap x down while smaller than child

PQ OPERATIONS (w/ HEAP):
- insert(x): O(logn) put x @ end, heapify_up(x)
- delete_max(): O(logn) swap root & last elt in array, delete last. heapify_down from root
- build: O(n) heapify every item down starting from leaves

SORTING:
- **HEAP SORT:** O(nlogn) S: ✗ I.P: ✓
  - build a heap (n = size of heap)
  - repeatedly remove max elt.

DAG SP: $O(|V|+|E|)$ acyclic + & - edge weights
- start distances from ∞
- get topological order w/ full DFS
- relax outgoing edges of each node in this order.
  - if $d[v] > d[u] + w(u,v)$, set $d[v] = d[u] + w(u,v)$ (triangle inequality)

BF & DAG SP - DIFFERENT METRICS:
- min, +: $(-\infty, \infty)$ → minimizes sum of edge weights
- min, ×: $[0, \infty)$

JOHNSONS:

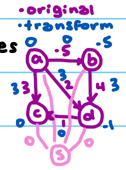
ALL PAIRS SHORTEST PATH: "distance b/t any 2"

GOAL: output a |V| × |V| table → $|E| \in \Omega(|V|^2)$
- if DAG: run DAG SP from each node $O(|V||E| + |V|^2)$
- if non-neg. weights: Dijkstras from each node $O(|V||E| + |V|^2 \log|V|)$

→ make weights non-negative (1) while preserving SP (2)?

① $\phi(v)$ is some function on V
for each v, subtract $\phi(v)$ from all incoming edges
add $\phi(v)$ to all outgoing edges
all paths from s to t change only by $+\phi(s) - \phi(t)$

② set $\phi(v)$ = SSSP $\delta(v)$ from some start node
new weight for (u,v): $w'(u,v) = w(u,v) + \delta(u) - \delta(s)$ (non-neg.)



1:
$\phi(a) = 0$
$\phi(b) = -5$
$\phi(c) = 0$
$\phi(d) = -1$

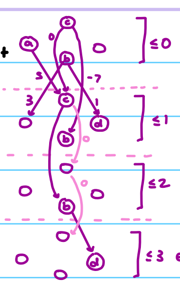
SSSP DISTANCE from ANY...: SUPERNODE
- add supernode w/ edges to all nodes. run BF from it. $O(|V||E|)$
- once reweighted, Dijkstras from every node. $O(|V||E| + |V|^2 \log|V|)$

*allows for disconnected nodes to be calc.

OUTPUT: min distance from every u→v in graph (often represented as matrix)

FLLOYD-WARSHALL: $O(|V|^3)$ - APSP
- DP alg. finding SP b/t all pairs of vertices in weighted graph
- undirected & directed, handles neg. edge weights
- doesn't handle neg. cycles

BELLMON-FORD: $O(|V| \cdot |E|)$ + & - weight
- any SP takes at most |V|-1 edges
- make |V| layers & run DAG SP to look at last layer distance
- "how far we can get after # edges"



NEGATIVE CYCLE DETECTION:
- BF & add another layer
- anything that has decr. at added layer is in negative cycle. can ID by tracing parent pointers
- DFS all points reachable to cycles & remove them.

HANDLING NEGATIVE CYCLES:
1) compute SCCs & condensation graph $O(|V|+|E|)$
2) within each SCC in original graph, run BF to learn which SCCs have negative cycles. $O(|V| \cdot |E|)$
3) weigh condensation graph edges s.t. outgoing edges from a negative cycle SCC have weight -1, o.w. edges have weight 0.
4) get DAG APSP in condensation graph $\delta_H(A,B)$
   - set $\delta_H(A,A) = -1$ if A contains neg. cycle
5) create G' by removing all negative SCCs
6) run Johnsons on G' to get $\delta'(u,v)$
7) go through every u,v let A,B be their SCCs
   if $\delta_H(A,B) < 0$, $\delta(u,v) = -\infty$
   o.w. $\delta(u,v) < \delta'(u,v)$

REC + PTS:

5. Given an unweighted graph $G = (V, E)$ in which some edges are red and some are blue, find a path from s to t with the minimal number of red edges.

Solution: We combine BFS and DFS. The main idea is that using a DEQUE, we can BFS w.r.t. red edges while *simultaneously* condensing blue edges using DFS.

BFS + DFS

- Initialize a DEQUE $Q = [s]$, $d(s) = 0$, and $P(s) = \perp$
- While Q is non-empty:
  - Remove the first element u from Q
  - For each unvisited out-neighbor v of u:
    - set $P(v) = u$
    - if (u, v) is red, set $d(v) = d(u) + 1$ and append v to Q
    - if (u, v) is blue, instead set $d(v) = d(u)$ and prepend v to Q
- Follow parent pointers from t and output the resulting path

Correctness and runtime proofs are as DFS and BFS.

3. There are n lock boxes and m keys. Each box has a distinct lock, so each key can open exactly one box. There's at least one copy of each box's key, but for some boxes there may be multiple copies of the key. Someone put all keys in the boxes and locked them up, but luckily they made a note of which keys are stored in each box. Keys and boxes are numbered so that we know which box is opened by each key. Some boxes contain no keys while others contain multiple keys. Boxes can also be forced open with a rusty crowbar. Design an algorithm to find the smallest set S of boxes that you need to force open in order to open all the other boxes. KS, CG + INDEGREES $O(n+m)$

Solution: Define a graph $G = (V, E)$ such that there is a node in V for each lock box and there is a directed edge $(u \to v) \in E$ between two lock boxes u and v if lock box u contains the key to lock box v.

The key idea is that if there is a path $a \to b \to c$ in G, then opening box a gives us the key to b, and opening box b gives us they key to c. So if we open a box, we get access to every box reachable from it in G. As a consequence, opening a box allows us to open all other boxes in its strongly connected component.

This last fact suggests we consider the condensation graph G_C. If a strongly connected component x of G is a *source* in G_C, meaning there are no edges into x, then we must force open some box in x. This gives as access to every box in x, as well as every box in every SCC reachable (in G_C) from x.

After forcing open one box in every source SCC, we're done: if a vertex in G_C isn't a source, we can follow edges backwards until reaching a source, so every vertex is 'downstream' of a source.

So this question is equivalent to finding the number of source SCCs in G_C. To do this, we construct G_C, and then for each edge $x \to y$ in G_C we mark y as not a source. The vertices that do not get marked are the sources. This takes $O(n + m)$ time.

1. Totodile has k PP to spend and wants to go to Tangelo Island. Design an $O(k|E|)$ time algorithm to decide whether Totodile can reach Tangelo Island. DUMMY NODES, BFS

Solution: This reduces to SPSP by using a variant of graph duplication that subdivides edges instead of duplicating the original vertices. We assume WLOG that G is connected; o/w we can prune unreachable vertices in $O(|E|)$ time using e.g. DFS.

- Construct a graph G_1 by subdividing every edge $\{u, v\}$ into a $c_{u,v}$-edge chain of edges if $c_{u,v} \leq k$ (instead remove the edge if $c_{u,v} > k$).
- BFS in G_1 from Shamouti Island to compute the distance d to Tangelo Island.
- Output $d \leq k$.

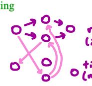
2. Ducklett has k PP to spend and wants to go to Tarroco Island. Ducklett can either swim across a route $\{u, v\}$ by spending $c_{u,v}$ PP, or fly over a route by spending 1 PP, regardless of $c_{u,v}$. However, after flying, Ducklett must rest and cannot fly again until after swimming across another route. Design an $O(k|E|)$ time algorithm to decide whether Ducklett can reach Tarroco Island. GRAPH DUPLICATION $\Theta(k|E|)$

Solution: To solve this problem, we combine the ideas of the previous part with some graph duplication ideas. We construct a G_2 with two vertices per island:

(a) vertex i_F corresponding to being at island i with Ducklett being rested

(b) vertex i_S corresponding to being at island i after flying

Then for each route $\{u, v\}$, add the following edges:



- a single directed edge from vertex u_F to v_S
- a single directed edge from vertex v_F to u_S

and if the cost of the route is at most k PP, we also add following additional edges:

- $c_{u,v}$-edge directed chain of edges from u_S to v_F
- $c_{u,v}$-edge directed chain of edges from v_S to u_F
- $c_{u,v}$-edge directed chain of edges from u_F to v_F
- $c_{u,v}$-edge directed chain of edges from v_F to u_F

The number of edges and vertices in this graph is still $\Theta(k|E|)$, so constructing this graph takes $\Theta(k|E|)$ time. Then any sequence of routes from Shamouti Island to Tarroco Island in this graph corresponds to visiting a sequence of islands without flying twice in a row. As before, we can run BFS from Shamouti Island (starting from the vertex that is allowed to fly to other islands) to find the distance d to Tarroco Island (either vertex) in $\Theta(k|E|)$ time. Ducklett can reach Tarroco Island iff $d \leq k$.

3. CIA officer Mary Cathison needs to drive to meet with an informant across an unwelcome city. Some roads in the city are equipped with government surveillance cameras, and Mary will be detained if cameras from more than one road observe her car on the way to her informant. Mary has a map describing the length of each road and knows which roads have surveillance cameras. Help Mary find the shortest drive to reach her informant, being seen by at most one surveillance camera along the way. DIJKSTRA + DUPLICATION

Solution: Construct a graph having two vertices $(v, 0)$ and $(v, 1)$ for every road intersection v within the city. Vertex (v, i) represents arriving at intersection v having already been spotted by exactly i camera(s). For each road from intersection u to v: add two directed edges from $(u, 0)$ to $(v, 0)$ and from $(u, 1)$ to $(v, 1)$ if traveling on the road will not be visible by a camera; and add one directed edge from $(u, 0)$ to $(v, 1)$ if traveling on the road will be visible. If s is Mary's start location and t is the location of the informant, any path from $(s, 0)$ to $(t, 0)$ or $(t, 1)$ in the constructed graph will be a path visible by at most one camera. Let n be the number of road intersections and m be the number of roads in the network. Assuming lengths of roads are positive, use Dijkstra's algorithm to find the shortest such path in $O(m + n \log n)$ time using a Fibonacci Heap for Dijkstra's priority queue.

4. Ash is trying to cycle from Pallet Town to Viridian City without destroying Misty's bike. For every trail e in the area, he knows the probability $p(e)$ of destroying the bike if he cycles along e. Help Ash find the safest path to Viridian City (the path that minimizes his probability of destroying the bike). Assume that all probabilities are independent and that arithmetic operations take constant time. DIJKSTRA, max ×, [0, 1]

Solution: Construct a graph having a vertex v for every trail intersection, and weight each edge (trail) e with $w(e) = 1 - p(e)$. Run Dijkstra using $(0, 1)$, max, $\times$, 1, 0 instead of $[0, \infty]$, min, $+$, 0, ∞, respectively. If n is the number of intersections and m the number of trails in the graph, then Dijkstra will take $O(m + n \log n)$ time using a Fibonacci Heap.

Problem 2. [15 points] **Tricky Tolls** (1 part)

A group of price-conscious MIT students is traveling from Syracuse to Toronto, and you wish to profit off of your fellow classmates. Your map shows that the students are considering traveling through n cities, connected by m two-way roads, and each road charges a toll of $1, except for the one road *you* own (between Hamilton and Burlington). You are allowed to set your toll price to be any *positive integer* you'd like.

Design an efficient algorithm that returns either the highest toll price you could set so that the MIT students will still choose to take your road, or state that no such toll exists. The students always pick the cheapest route. In case of a tie, the students will pick the route that does not use your road. Analyze the runtime of your algorithm. You don't need to prove correctness. ONE UNIQUE EDGE / 2 BFS

Solution: Build the (unweighted, undirected) graph G without your road. Output

$$\max \begin{Bmatrix} \delta(s,t) - \delta(s,h) - \delta(b,t) - 1 \\ \delta(s,t) - \delta(s,b) - \delta(h,t) - 1 \end{Bmatrix} \text{ if it is positive, else } \perp.$$

Here, $\delta(s, \bullet)$ can be computed using a BFS on G from s, and $\delta(\bullet, t)$ can be computed using a BFS on G_{rev} from t.

• BFS of G w/o road → $\delta_1(s,t)$
• BFS of G w/ road → $\delta_2(s,t)$

Solution: Build the graph G without your road, and G' with your road. Output

$$\delta_G(s,t) - \delta_{G'}(s,t) \text{ if it is positive, else } \perp,$$

output $\delta_1 - \delta_2$ (how much we can change)

after running a BFS from s on G and on G'.

Let $G = (V, E)$ be a directed graph. Say that vertices $u, v \in V$ are *semi-connected* iff G contains either a u-v path or a v-u path (or both).

(a) [12 points] Suppose G is acyclic. Describe an algorithm that finds and outputs a pair of vertices that are **not** semi-connected, or $\perp$ if no such pair exists. Briefly justify correctness. Prove **one** of the runtime upper bounds below. Mark your selected option by filling in the square completely.

■ For up to 12 points, prove that your algorithm runs in $O(|V| + |E|)$ time.

☐ For up to 10 points, prove that your algorithm runs in $O(|V| \log |V| + |E|)$ time.

☐ For up to 6 points, prove that your algorithm runs in $O(|V||E|)$ time.

Solution: 12pts TOPOLOGICAL SORT

1. Sort vertices topologically.
2. Output any two consecutive non-adjacent vertices, or $\perp$ if no such pair exists.

Runtime Analysis:
Step 1 takes $\Theta(|V| + |E|)$ time with Full DFS. Step 2 takes $\Theta(|V|)$ time. Total is $\Theta(|V| + |E|)$.

Correctness:
Suppose u and v are output. Then there is no path of length ≤ 1 between u and v by construction. Any longer path would have to pass through a vertex that is not between u and v in the topological order, so would have an edge that violates the order.

Suppose instead that $\perp$ is output. Then the topological order is a path of length $n - 1$, i.e. all vertices are semi-connected.

Problem 4. [16 points] **Prismatic Paths** (1 part)

Let $G = (V, E, w)$ be a positive-weighted, undirected graph, and let $s, t \in V$. Suppose each edge of G has one of seven colors, and every vertex has exactly one incident edge of each color. Design an $O(|V| \log |V|)$ algorithm to compute the shortest s-t path that contains edges of at least three different colors. Analyze your algorithm's runtime and briefly justify correctness. DUPLICATION

Solution: We duplicate G 30 times to keep track of the colors seen so far along a path: the 29 sets of at most two colors, plus a success state $\star$ that collapses all sets of at least three different colors. (Note: this could instead be done with 2^7 states and a supersink.) state = (v, {#}), (v, {#, #}), (v, {#3}) how many colors seen

1. Construct a directed graph $G' = (V', E', w')$, where:
   - For every vertex $v \in V$ and every color set S relevant to v (described above), V' contains the vertex (v, S). ≤2: 0 col (1) + 1 col (7) + 2 col (21) = 29
   - For every (v, S) in V' and every edge $\{u, v\} \in E$ of color c, E' contains an edge from (v, S) to $(u, S \cup \{c\})$ with the same weight as $\{u, v\}$ — 7 choose 2
2. Run Dijkstra from $(s, \emptyset)$ to $(t, \star)$ and output its projection onto G

Runtime: (1) takes linear time, as G' has $30|V|$ vertices and $210|V|$ edges. (2) takes time $O(|V| \log |V| + |V|)$ because G' is sparse and has size $O(|V|)$. Total is $O(|V| \log |V|)$.

Problem 6. [18 points] **Single Source Second-Shortest Paths** (1 part)

Let $G = (V, E, w)$ be a directed graph with non-negative edge weights. For vertices $s, t \in V$, define the **second-shortest distance** $\Delta(s, t)$ to be minimal such that:

- $\Delta(s, t) > \delta(s, t)$
- $\Delta(s, t)$ is the length of some s-t path REWEIGHT, DIJKSTRAS

If every s-t path is a shortest path (and in particular, if $\delta(s, t) = \infty$), then $\Delta(s, t) = \infty$. Design an algorithm that takes as input G and vertices s and t, and computes $\Delta(s, t)$. Analyze your algorithm's runtime. For full credit, it should run in time $O(|V| \log |V| + |E|)$. You need not prove correctness.
Hint: How can you track whether a path has deviated from all shortest paths?

Solution:

1. Compute SSSP in G from s, denoted $\delta_G(s, -)$
2. Reweight all edges using potential $\delta_G(s, -)$
3. Duplicate G to create graph $G' = (V', E')$ with two layers
   - V' contains two copies v_0 and v_1 of every vertex $v \in V$
     - v_0 indicates that v has been reached via a shortest path
     - v_1 indicates that v has been reached via a non-shortest path
   - Edges $(u, v) \in E$ lift to E'
     - (u_0, v_0) and (u_1, v_1) are always included
     - (u_0, v_1) is also included if (u, v) has positive weight
4. Compute SSSP from s_0 in G', denoted $\delta_{G'}(s, -)$, and output $\delta_G(s, t) + \delta_{G'}(s_0, t_1)$

Runtime: (1) takes $O(|V| \log |V| + |E|)$ time with Dijkstra. (2) takes $O(|E|)$ time. (3) takes linear time. (4) takes $O(|V| \log |V| + |E|)$ time with Dijkstra, as G' has $2|V|$ vertices and $3|E|$ edges. Total is $O(|V| \log |V| + |E|)$.

will bike to the meeting location from their home along dirt roads. Each road segment between intersections has a level of **fun** associated with biking along it in a certain direction. Road fun-ness may be positive, but could also be be negative, e.g., when a road is difficult to traverse in a given direction, or passes by a scary dog, etc. BF + MAXIMIZE (-1)

The children would like to maximize their total fun, which accumulates additively over the road segments they individually bike over. Help the children plan their day by finding an optimal tee-ball location, or return a continuously-fun bike loop in their district, if one exists. You may assume that each child can reach any road in the district by bike.

Solution: Construct a graph on road intersections within the district, as well as the locations a, b, and c of the homes of the three children, with a directed edge from one vertex to another if there is a road between them traversable in that direction by bike. Weight the edge by the negative fun-ness of the road. If a negative-weight cycle exists in this graph, such a cycle would represent a continuously fun bike loop. To check for the existence of any negative-weight cycle in the graph, run Bellman-Ford from vertex a. If Bellman-Ford detects a negative-weight cycle by finding an edge (u, v) that can be relaxed in round $|V|$, trace the parent pointers back from v, until a cycle is found, and return the roads in that cycle as a continuously-fun bike loop.

Alternatively, if no negative-weight cycle exists, minimal-weighted paths correspond to bike routes that maximize fun. Our Bellman-Ford from vertex a thus computed shortest paths $\delta(a, v)$ from a to each vertex v in the graph. Run Bellman-Ford two more times, once from vertex b and once from vertex c, computing shortest paths values $\delta(b, v)$ and $\delta(c, v)$, respectively, for each vertex v in the graph. Then for each vertex v, compute the sum $\delta(a, v) + \delta(b, v) + \delta(c, v)$. A vertex that minimizes this sum will correspond to a road intersection that maximizes total fun of all three children in reaching it.

This algorithm runs Bellman-Ford three times and then compares a constant-sized sum at each vertex, so this algorithm runs in $O(|V||E|)$ time.

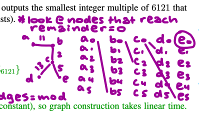
Problem 5. [30 points] **Mod** (2 parts) GRAPH DUPLICATION (CONST)

In each of the following, let $G = (V, E)$ be an undirected graph with integer weights, and let $s, t \in V$. You must analyze your runtime, but you need not prove correctness.

(a) [15 points] Give an $O(|V||E|)$ time algorithm that outputs the smallest integer multiple of 6121 that is the weight of some s-t path (or state that none exists). • look e nodes that reach remainder=0

Solution:

- Create $G' = (V', E')$, where
  - $V' = V \times \mathbb{Z}_{6121}$
  - $E' = \{(u_z, v_{z+w(u,v)}) : (u, v) \in E, z \in \mathbb{Z}_{6121}\}$
  - $w'(u_z, v_y) = w(u, v)$
- BF in G' to compute and output $\delta(s_0, t_0)$



Runtime: G' is larger than G by a factor of 6121 (constant), so graph construction takes linear time. BF runtime is still $O(|V||E|)$. Total is $O(|V||E|)$.

Correctness (not required) follows from the 1-1 correspondence between s-v paths of weight x in G and s_0-$v_{\text{rem}(x,6121)}$ paths of length x in G'.

Solution:

- DFS from s in G
- If t is not reachable, output ∞
- If any negative edge is reachable, output $-\infty$
- Let G' be as above, and use Dijkstra to compute and output $\delta(s_0, t_0)$

Runtime: Graph construction is as above. DFS takes linear time. Dijkstra takes $O(|V| \log |V| + |E|)$, which dominates.

Correctness (not required) hinges on correctness of third step. Suppose G contains a negative edge (u, v) reachable from s. Let π_1 be a path from s to u, and π_2 be a path from u to t. Let $k = \max(1, w(\pi_1 \pi_2))$. Then $\pi := \pi_1 [u, v, u]^{2k} \pi_2$ is a negative s-t path, and $(\pi \pi^{-1})^{6121}$ lifts to a negative cycle through s_0 in G', and $(\pi \pi^{-1})^{3060} \pi$ lifts to a path from s_0 to t_0.

Problem 5. [12 points] **The Lectourist** (2 parts)

Erik wants to deliver his lecture "Puzzling with Paper" in Tokyo. He has a list of n cities he can potentially travel to. There are m flights between cities, and a flight from city u to another city v costs $p_{u,v}$ dollars. When Erik is in city v, he can earn r_v dollars by delivering his lecture there. All values of $p_{u,v}$ and r_v are non-negative. Erik cannot give two consecutive lectures in the same city. Assume that flights are expensive enough that Erik cannot make an infinite amount of money.

(a) [9 points] Design an algorithm that finds a tour path from Boston to Tokyo that minimizes Erik's net cost. Analyze your algorithm's runtime. For full credit, your algorithm should be as efficient as possible. You need not prove correctness. REWEIGHT, BF

Solution:

1. Create a graph $G = (V, E)$:
   - V is the set of cities
   - E contains an edge from u to v of weight $p_{u,v} - r_v$ iff there is a flight from u to v
2. B-F from Boston and output the SP to Tokyo

Runtime: Assuming all cities are represented as integers in the range $[1, n]$, step (1) takes $O(n + m)$ time. Step (2) takes time $O(nm)$, as G has n vertices and m edges. Total is $O(nm)$.
Note: assumption can be enforced in expected linear time with a hash table. As long as we use airport codes (or at least avoid Bangkok), it's probably safe to assume that the input size is at most $O(nm)$.

Problem 3. **Hide and Seek** SKITTY

Hoothoot and Skitty are playing Hide and Seek on a directed graph $G = (V, E)$ in which every vertex has a self-loop. At time 0, Hoothoot and Skitty start at vertices v_H and v_S respectively. Hoothoot gets a head start of k moves. Hoothoot chooses a movement function $f_H \subseteq E$, i.e. for every vertex v, $f_H(v)$ is an out-neighbor of v. Skitty then (knowing f_H) chooses a movement function $f_S \subseteq E$. At time $t \in \mathbb{N}$, Hoothoot moves to $f_H^t(v_H)$. If $t < k$, then Skitty stays at v_S, and if $t \geq k$, then Skitty moves to $f_S^{t-k}(v_S)$. Skitty wins when the two Pokémon are at the same vertex.
Note: $f^i(v)$ is defined inductively by $f^0(v) := v$, and $f^{i+1}(v) := f(f^i(v))$.

(a) Hoothoot wants to evade Skitty for as long as possible. A naïve attempt is to compute the vertex x that is reachable from v_H and as far from v_S as possible, go to x, and wait. Describe an example (G, v_H, v_S, k) for which this naïve strategy does *not* work. Briefly justify your answer.

Solution: Let G be a bidirectional line of 4 vertices, let v_H be the leftmost vertex, and let v_S be its neighbor. Then Hoothoot's best strategy is to stay put (so that Skitty wins at time $k + 1$), whereas the naïve strategy causes Hoothoot to throw immediately at time 1.

(b) Help Hoothoot design a linear time algorithm to evade Skitty for as long as possible, given G, v_H, v_S, and k as input. Your algorithm should output f_H that maximizes the time T such that Skitty cannot win before time T regardless of f_S. Prove correctness. You need not analyze runtime.

Solution:

1. Use BFS to compute SSSP from v_S in G; record these distances as δ_S
2. Add k to all of the above distances except $\delta_S(v_S)$
3. Use a modified BFS to compute distances δ_H from v_H in G
   - When relaxing an edge $\{a, b\}$, if $\delta_H(a) + 1 \geq \delta_S(b)$, disregard $\{a, b\}$
4. Find x with finite $\delta_H(x)$ that maximizes $\delta_S(x)$
5. Hoothoot should take the shortest path to x as computed from δ_H and then wait

Correctness: By construction, Skitty cannot win before Hoothoot reaches x. Skitty can reach v (and win) at time $\delta_S(x)$. This value is maximized by construction.

(c) Help Skitty design a linear time algorithm to compute f_S, given G, v_H, v_S, k, and f_H as input. Your algorithm should output f_S that minimizes the time at which Skitty wins, or output $\perp$ if Skitty cannot win. Analyze your algorithm's runtime and prove correctness.
Hint: $\Omega(k)$ is too slow, but Euclidean division is not!

Solution:

1. Compute Hoothoot's locations $f_H^t(v_H)$ iteratively and store them in a DAA, up to the first time he revisits a vertex. If the first duplicated vertex is reached at times t_1, t_2, we find locations up until $f_H^{t_2}(v_H)$.

PSETS:

Problem 1. [30 points] **Liza Pover** ← PSET 1

The Stata Center is a large labyrinth full of food protected by the CSAIL[1], LIDS[2], and DLP[3] research groups. Many brave the labyrinth in search of free food, but few escape.

Liza Pover is on a quest for free pizza. She has downloaded a map from ⌊plans⌋.mit.edu consisting of a set L of **locations** (including rooms, hallways, staircases, and elevators) and a set D of **doors** that connect pairs of locations. Each door $d = (\{\ell_1, \ell_2\}, t)$ connects two locations ℓ_1 and ℓ_2 and has a lock of type $t \in \{\text{CSAIL}, \text{LIDS}, \text{DLP}, \text{MIT}\}$. Anyone with a matching key can pass the door in either direction. For each key type t, Liza knows a set of locations $L_t \subseteq L$ where she can find a key of type t. She also knows a set of locations $\Pi \subseteq L$ where she can find a pizza.

Liza is at an entrance location s and has an MIT key. Her goal is to find (at least) one pizza and then return to s (not necessarily by the same path) as quickly as possible.

(a) Describe a linear time algorithm to find a path that Liza can take from s to s that collects at least one pizza and minimizes the number of times Liza must pass a door (or $\perp$ if no such path exists). Briefly justify correctness, and analyze runtime.

Solution: While Liza explores Stata, the doors she can use depend on which keys she has in her possession. It would be useful to know which keys she has while exploring the building. There are 4 types of keys, but she will always possess an MIT key. Thus, there are at most 2^3 possible sets of keys she could have at any given time, to account for any subset of {CSAIL, LIDS, DLP} she could have. We will use graph duplication to represent these $2^3 \in O(1)$ possible states. DUPLICATION + BFS

Construct a graph $G = (V, E)$ as follows.

- For each location $\ell \in L$, construct 2^4 vertices $(\ell, p, k_{\text{CSAIL}}, k_{\text{LIDS}}, k_{\text{DLP}})$ where p and k_t are Boolean. This represents Liza being in location ℓ with p pizzas, and k_t keys of type t for each $t \in T$.
- For each door $(\{\ell_1, \ell_2\}, t)$, and each vertex $v = (\ell_1, p, k_{\text{CSAIL}}, k_{\text{LIDS}}, k_{\text{DLP}})$ with $k_t = 1$, add a directed edge from v to $(\ell_2, p', k'_{\text{CSAIL}}, k'_{\text{LIDS}}, k'_{\text{DLP}})$, where $p' = p \vee (\ell_2 \in \Pi)$ and $k'_t = k_t \vee (\ell_2 \in L_t)$.

This graph exactly encodes all possible state transitions from any location in the Stata Center while keeping track of the number of pizzas and key cards that Liza can be holding. Thus a path from vertex $u = (s, s \in \Pi, s \in L_{\text{CSAIL}}, s \in L_{\text{LIDS}}, s \in L_{\text{DLP}})$ to any vertex $v \in Q = \{(s, 1, k_{\text{CSAIL}}, k_{\text{LIDS}}, k_{\text{DLP}})\}$ represents a path that starts and ends at s, while also procuring a pizza. Since Liza would like to minimize the number of doors she must cross, running BFS in G from u to each $v \in Q$ finds the minimum doors crossed to each of them, so we can return a shortest path to any of them by traversing parent pointers back to the source.

Graph G has $|L| \cdot 2^4 = O(|L|)$ vertices and at most $|D| \cdot 2^4 = O(|D|)$ edges, so can be constructed in $O(|L| + |D|)$ time, and running BFS once from s also takes time linear in the size of the graph, so this algorithm runs in $O(|L| + |D|)$ time.

(b) Liza fails to acquire a pizza, but she does escape. She returns with k lock picks. Liza can use a lock pick to open a locked door for which she doesn't yet have the correct key. This breaks the lock pick, and the door locks again behind her. Describe an $O(k|L| + k|D|)$ time algorithm to find a path that Liza can take from s to s that collects at least one pizza and minimizes the number of times Liza must pass a door (or $\perp$ if no such path exists). Briefly justify correctness, and analyze runtime.

Solution: Modify the solution above to track the number of lock picks Liza has as part of the state. TRACK STATE

- Vertex set is $V = L \times \{0, 1\}^4 \times \{0, 1, 2, \ldots, k\}$, where the last component is the number of remaining lock picks in Liza's possession.
- For each door $(\{\ell_1, \ell_2\}, t)$, and each vertex $v = (\ell_1, p, k_{\text{CSAIL}}, k_{\text{LIDS}}, k_{\text{DLP}}, x)$ with $x \geq 0$, add an edge from v to $(\ell_2, p', k'_{\text{CSAIL}}, k'_{\text{LIDS}}, k'_{\text{DLP}}, x - 1 + k_t)$ if it exists, where $p' = p \vee (\ell_2 \in \Pi)$ and $k'_t = k_t \vee (\ell_2 \in L_t)$. This allows Liza to move to ℓ_2 using a key if she has it, or failing that, a lock pick if she still has one.
- Source is $u = (s, s \in \Pi, s \in L_{\text{CSAIL}}, s \in L_{\text{LIDS}}, s \in L_{\text{DLP}}, k)$, and target set is $Q = \{(s, 1, k_{\text{CSAIL}}, k_{\text{LIDS}}, k_{\text{DLP}}, x)\}$
- Vertex set and edge set are larger by a factor of $O(k)$, so runtime is increased by a factor of $O(k)$.

(a) Your advisor thinks that Course 19 is a waste of time, so demands that you take only one class from Course 19 per semester. You would rather not pay tuition indefinitely, so design a linear time algorithm that outputs a valid schedule of Course 19 classes (with only one class per semester) that minimizes the number of semesters you need before taking 19.1434. Analyze its runtime, and briefly justify correctness.

Solution: Construct a graph $G = (V, E)$, where V is the set of Course 19 classes, and E contains an edge from u to v iff v is a prerequisite for u. DFS from IHTFP, and take all classes with finite finish times in increasing order of finish time (unless a cycle is detected, in which case no schedule exists).

Runtime: Constructing G takes linear time by definition. DFS takes linear time.

Correctness: Reachable classes are the (direct or indirect) prerequisites for IHTFP. G is the reverse prerequisite graph, so a topological order gives a reversed schedule. DFS finishing order topo sorts in reverse. REVERSE + TOPO

(b) You ignore your advisor. Design a linear time algorithm that outputs a valid schedule of Course 19 classes (with no limit on the number of classes each semester) that minimizes the number of semesters you need before taking 19.1434. Analyze its runtime, and briefly justify correctness.

Solution: • count levels of G

1. Construct G as before
2. Define a (partial) function $f : |V| \to \mathbb{N}$
3. For every class v in the order given by part (a):
   - Set $f(v)$ to be the max over all out-neighbors u of v of $1 + f(u)$
   - If v has no out-neighbors, set $f(v) = 0$
4. Output f, i.e. for every class v in the schedule, take v in semester $f(v)$

Runtime: Step 1 takes linear time, step 3 iterates over every vertex and every edge once so takes linear time, and steps 2 and 4 take linear time for any reasonable implementation of f.

Correctness: We maintain the invariant that f is a valid schedule, and for every v for which $f(v)$ has been set, $f(v)$ is the earliest semester in which v can be taken. This is vacuously true at the start. At the iteration in which $f(v)$ is set, we know from (a) that if u is a prerequisite for v, then $f(u)$ has already been set. By the invariant, v cannot be taken at or before $f(u)$ for any such u. However, f allows you to take all prerequisites for v at or before semester $\max f(u)$, so v can be taken the following

(c) Help Skitty design a linear time algorithm to compute f_S, given G, v_H, v_S, k, and f_H as input. Your algorithm should output f_S that minimizes the time at which Skitty wins, or output $\perp$ if Skitty cannot win. Analyze your algorithm's runtime and prove correctness.
Hint: $\Omega(k)$ is too slow, but Euclidean division is not!

Solution:

1. Compute Hoothoot's locations $f_H^t(v_H)$ iteratively and store them in a DAA, up to the first time he revisits a vertex. If the first duplicated vertex is reached at times t_1, t_2, we find locations up until $f_H^{t_2}(v_H)$.
2. Compute $k' := \min(k, t_1 + \text{rem}(k - t_1, t_2 - t_1))$
3. Use BFS to compute SSSP from v_S in G
4. Compute Hoothoot's locations up to time $k' + |V|$
5. Find the first t such that $\delta_S(f_H^t(v_H)) + k' \leq t$
6. Skitty should take the shortest path to $f_H^t(v_H)$ (if one exists) and then wait

Hoot Hoot repeating vertices - can mod
*Skitty waits until Hoot Hoot's movements are predictable

Runtime analysis:

1. Takes linear time by pigeonhole
2. Takes linear time with long division
3. Takes linear time
4. Takes linear time by construction
5. This is a linear scan
6. A function on $|V|$ takes linear time to output

Total is linear.

Correctness: Hoothoot will repeat the loop from t_1 to t_2 forever. Thus, if $k > t_1$, we may reduce k modulo $t_2 - t_1$ to something in the range $[t_1, t_2)$ without changing the outcome. We now assume WLOG $k = k'$. LHS of step 5 is the first time Skitty can reach $f_H^t(v_H)$. If the condition is satisfied, then Skitty can win at time t using the g from step 6. Otherwise, Skitty cannot win at time t. Finally, we observe that finite distances are bounded by $|V|$, so Skitty can win by time $k + |V|$ if at all.

Problem 1. **Magikarp Jump** ← PSET 2

Maggie is running a Magikarp Jump contest. In each round, all n Magikarps jump once, and each jump height is recorded. After each round, Maggie grabs a bag of berries, and allows the highest-jumping Magikarps to choose one berry each, in decreasing order of jump height.

Help Maggie find an algorithm that, given an array of n jump heights and the number k of berries in Maggie's bag, outputs the order in which the highest-jumping Magikarps should choose the berries.

(a) Prove that Maggie's task must take $\Omega(n + k \log n)$ time in the comparison model.

Solution: We condition on whether $k \leq \frac{n}{\log n}$.

- Case 1 $\left(k \leq \frac{n}{\log n}\right)$: In this case, $k \log n \leq n$, so $\Omega(n + k \log n) = \Omega(n)$. We indeed require $\Omega(n)$ time just to read the input and find the single best Magikarp.
- Case 2 $\left(k > \frac{n}{\log n}\right)$: In this case, $\Omega(n + k \log n) = \Omega(k \log n)$. Since $\log k > \log n - \log \log n \in \Omega(\log n)$, $\Omega(k \log n) \subseteq \Omega(k \log k)$. Indeed, we must sort at least k values, which requires $\Omega(k \log k)$ time by the lower bound from class.

(b) Describe an asymptotically optimal algorithm for this task, and analyze its runtime.

Solution: Since we have lower bounded the complexity of this problem as $\Omega(n + k \log n)$, it suffices to construct an algorithm and prove an $O(n + k \log n)$ time upper bound.

Algorithm Description MIN HEAP

We build a min heap out of the input elements $(A[i], i)$, keyed lexicographically. Next, we pop from this heap k times and put the removed indices (in order) into an output array of size k.

Runtime Analysis

We learned in class that building the heap takes $O(n)$ time, and that popping from the heap takes $O(\log n)$ time. Since we build the heap once, and pop k times, the total runtime is

$$O(n + n + k \log n) = O(n + k \log n).$$

(a) Suppose Inteleon knows that the proximity fuses have range k, and he wants to complete his mission as quickly as possible. Describe an $O(n \log n + m)$ algorithm to find a shortest k-safe path, or output $\perp$ if no such path exists. Briefly (2-3 sentences) justify correctness. You need not analyze runtime.

Solution: DIJKSTRA + SUPERNODE

1. Create a weighted undirected graph $G = (V, E)$, where V is the set of intersections, and E contains an edge $\{u, v\}$ with weight ℓ iff there is a road of length ℓ between u and v.
2. Add a supersource u connected to each mine with an edge of weight 0.
3. Run Dijkstra from u to find $\delta(u, v)$ for each vertex v.
4. Delete any v where $\delta(u, v) < k$.
5. Run Dijkstra from s to t and output the result.

Correctness: Since u has distance 0 from every mine, each vertex v's distance from u is also its distance from the closest mine. Deleting vertices that are too close to mines leaves exactly the vertices that could be on k-safe paths.

(b) Suppose Inteleon does not know the range of the proximity fuses, so he prioritizes safety over speed. Describe an $O(n \log n + m)$ algorithm to find a shortest k-safe path, where k is maximal such that a k-safe path exists. If no 0-safe path exists, output $\perp$. Briefly analyze runtime. You need not justify correctness.

Solution: • maximize min edge safety (Dijk max-min) • remove vertices with < k safety

1. As above
2. As above
3. As above
4. Reweight each edge $\{v, w\}$ to $\min(\delta(u, v), \delta(u, w))$.
5. Run Dijkstra from s using max-min metric to find $\delta'(s, v)$ for each vertex v.
6. Delete all vertices v with $\delta'(s, v) < \delta'(s, t)$.
7. Reweight back to original weights.
8. Run Dijkstra from s to t using the min-plus metric and output the result.

Runtime: Creating G takes linear time, and the modifications don't increase the size of G by more than a constant factor. Dijkstra's runtime is asymptotically unaffected, so the algorithm's runtime is dominated by the first $O(n \log n + m)$ Dijkstra call.

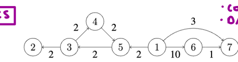
Problem 1. **Smell The Roses**

Roselia is not impressed with the shortest path algorithms she is learning in 6.1210. She would like to find longest paths instead, so she can spend as long as possible enjoying the scenery on her way to her destinations. There are n locations numbered 1 to n in Roselia's garden, connected by m one-way trails of positive length. Roselia's home is located at location s. Give a linear time algorithm to find the lengths of the **longest paths** from Roselia's home to each location in the garden.

Note that the paths need not be simple. This means that for some destinations, the longest path may have infinite length. In this case, you should still find the longest path for any destinations with longest paths from s of finite length. For example, in the garden below, if $s = 1$, your algorithm should find longest paths from 1 to 6 and 7 even though nodes 2, 3, 4, 5 have infinite longest paths. If the longest path to a node is infinite, your algorithm should return ∞ for that node. If a node is unreachable from s, you should return $-\infty$.

• find reachable
• condense SCCs
• DAG relax

DFS + SCCS



Solution: First, perform a DFS from s to get all nodes reachable from s. The longest path for any non-reachable nodes is non-existent, so mark the longest path as $-\infty$ and remove these nodes and their corresponding edges from the graph. Call $G = (V, E)$ the remaining graph of nodes reachable from s.

When finding the longest path from s to t, if there is an SCC C of size > 1 between s and t such that you can reach C from s and you can reach t from C, the max distance is infinite. This is because we can just keep hopping from node to node within the SCC, continually increasing the path length. Therefore, any node within an SCC of size > 1 or that is reachable from an SCC of size > 1 has an infinite length longest path and should be removed from consideration.

In order to do this we use Kosaraju's to find the SCCs in $O(m+n)$. We then build the condensation graph $G^c = (V^c, E^c)$ of SCCs, where each node corresponds to an SCC and edge $(U, V) \in E^c$ only if there exists $u \in U, v \in V$ such that $(u, v) \in E$. We connect a supernode S to all nodes corresponding to SCCs of size > 1 and BFS (or DFS) from S in $O(m+n)$ to find all reachable SCCs. We then remove any node in an SCC reachable from S from G and mark the longest path to these nodes as infinite. Call the remaining graph G'.

At this point, the remaining graph G' must be acyclic – any nodes that were in a cycle would have been removed in the previous step. Thus, if we negate all of the weights in G', run DAG relaxation from s, and re-negate the answers, we will compute the longest path for any remaining nodes.

Each step of this algorithm (DFS, Kosaraju's, DAG Relaxation) takes $O(m+n)$ time, so the total time is $O(m+n)$.

Purrloin is exploring the Unova Region. There are n cities, and there are $O(n)$ one-way roads between pairs of cities that Purrloin can travel. Every city can be reached from every other city by traveling along roads.

Purrloin must pay $c(u, v)$ coins to travel along the road between cities u and v. Note that this cost may be positive (if there are tolls) or negative (if Purrloin can purloin[1] enough coins to offset the tolls). Some cities are designated as Pokémon Centers, and Purrloin is currently at the Nimbasa City Pokémon Center. Purrloin must start each day at a Pokémon Center, and cannot lose (meaning net loss) more than k coins in a single day. You may assume that Purrloin currently has arbitrarily many coins but cannot pilfer[1] infinitely many coins.

(a) Purrloin wants to explore as many cities as possible (possibly over many days) and then end up back at the Nimbasa City Pokémon Center. Describe an $O(n^2 \log n)$ algorithm to find all cities that Purrloin can explore. Briefly analyze its runtime and justify correctness.

Solution: JOHNSONS • city is valid if $d(PC, x) + d(x, PC) \leq k$ for PC reachable from Nim.

1. Create a graph $G = (V, E)$:
   - V is the set of cities.
   - E is the set of pairs (u, v) such that there is a road from u to v.
2. Run Johnson's on (G, c) to compute δ.
3. Create a graph $G' = (V', E')$:
   - V' is the set of cities designated as Pokémon Centers.
   - For each pair of vertices (u, v) in V', E' contains edge (u, v) iff $\delta(u, v) \leq k$.
4. Let S be the SCC of the Nimbasa City Pokémon Center in G'.
5. Purrloin can visit a city v iff $\min_{s \in S} \delta(s, v) + \min_{s \in S} \delta(v, s) \leq k$.

Proof of correctness: G gives a natural correspondence between cities and roads, and edges, and costs and weights. G' is a graph of Pokémon Centers, with edges representing allowed paths that Purrloin can spend the night, while still retaining the ability to return to the Nimbasa City Pokémon Center at the end of a day. S is then the set of all Pokémon Centers where Purrloin can spend the night. A city v can be reached iff the shortest path between cities in S through v has total cost at most k. This shortest path is a concatenation of the shortest paths from S to v and from v to S.

Runtime analysis: Step 1 takes $O(n+m) = O(n)$ time since $m \in O(n)$ as given by the problem statement. G has n vertices and m edges, so step 2 takes $O(n^2 \log n + mn) = O(n^2 \log n)$ time. G' contains at most n vertices and at most n^2 edges, so can be constructed in $O(n^2)$ time. Step 4 takes $O(n^2)$ time with e.g. Kosaraju-Sharir. Step 5 iterates over $V \times V'$, so takes $O(n^2)$ time. Total runtime is dominated by step 2, so is $O(n^2 \log n)$.

(b) Purrloin is now planning to pinch[1] something from Nuvema Town. Describe an $O(n^3)$ algorithm to find the cost of a cheapest path that Purrloin can take from the Nimbasa City Pokémon Center to Nuvema Town and back. Briefly analyze its runtime and prove correctness.

Solution: BF

1. Let G, G', and δ be as above.
2. Run Bellman-Ford from Nimbasa City on (G', δ) to compute all distances δ' from Nimbasa City.
3. Reverse all edges and run Bellman-Ford again to compute all distances δ'' to

Problem 3. [30 points] **Holiday Migration** (2 parts)

Delibird is on her annual holiday migration from Boston to Miami while delivering presents along the U.S. East Coast. There are n cities along the coast, indexed $1, \ldots, n$ from north to south. City 1 is Boston and city n is Miami. Delibird can fly south by at most k cities at a time. This means if she's in city i, she can fly to city $i + j$ as long as $1 \leq j \leq k$ and $i + j \leq n$.

Delibird starts in Boston with 0 presents. Whenever she leaves any city, she gives away half of her presents (rounded up). Whenever she arrives in city i, she acquires p_i new presents. All p_i are non-negative integers.

(a) [15 points] Design an $O(nk)$ time DP to determine whether or not it is possible for Delibird to end her trip (in Miami) with at least 12 presents. Correct $O(n^2)$ algorithms will be eligible for partial credit.

You must use one of the two subproblem definitions given below. Clearly indicate which subproblem you use. Both definitions can lead to correct solutions.

■ $X(i)$ (where $1 \leq i \leq n$) is the maximum number of presents Delibird can have after traveling from city 1 to i.

■ $Y(i)$ (where $1 \leq i \leq n$) is the minimum number of presents Delibird must have while in city i, so that after traveling from city i to n she will have at least 12 presents.

Solution:

S X
R $X(i) = \max_j \{\lfloor X(j)/2 \rfloor + p_i \mid \max(1, i-k) \leq j < i\}$
T Each $X(i)$ depends on $X(j)$ with $j < i$
B $X(1) = 0$
O $1[X(n) \geq 12]$
T There are $O(n)$ subproblems, and each takes $O(k)$ to evaluate by iterating over j, for a total of $O(nk)$ runtime.

OR

S Y
R $Y(i) = \max\left\{0, \min_j \{2(Y(j) - p_j) \mid i < j \leq \min(n, i+k)\}\right\}$
T Each $Y(i)$ depends on $Y(j)$ with $j > i$
B $Y(n) = 12$
O $1[Y(1) \leq 0]$
T There are $O(n)$ subproblems, and each takes $O(k)$ to evaluate by iterating over j, for a total of $O(nk)$ runtime.

Note that the following relate statements are equivalent to the above solutions and would also earn full credit:

- $X(i) = \max_j \{\lfloor X(i-j)/2 \rfloor + p_i \mid 1 \leq j \leq k, i - j \geq 1\}$
- $Y(i) = \max\left\{0, \min_j \{2(Y(i+j) - p_{i+j}) \mid 1 \leq j \leq k, i + j \leq n\}\right\}$

CATTU
6.121 1

KARATSUBA:

A = multiply (x_{lo}, y_{lo})
B = multiply (x_{hi}, y_{hi})
E = multiply ($x_{lo}+x_{hi}$, $y_{lo}+y_{hi}$)
B + C = E − A − D
$T(n) = 3T(\frac{n}{2}) + C_2 n$
$\Theta(n^{\log_2 3})$

$\log < x^n < n^x < n!$
poly exp fact

MASTER'S THM:

let $T(n) = aT(\lceil \frac{n}{b} \rceil) + f(n)$, $b > 1$, $a \geq 1$

- O(n) work done @ root ≤ CASE 1: if $f(n) \in O(n^{(\log_b a) - \varepsilon})$, for some $\varepsilon > 0$, $T(n) \in \Theta(n^{\log_b a})$
- O(nlogn) evenly dist. = CASE 2: if $f(n) \in \Theta(n^{\log_b a})$, $T(n) \in \Theta(n^{\log_b a} \cdot \log n)$
- O(n²) work done @ leaves ≥ CASE 3: if $f(n) \in \Omega(n^{(\log_b a) + \varepsilon})$ & $af(\lceil \frac{n}{b} \rceil) \leq c f(n)$, $T(n) \in \Theta(f(n))$ for some $\varepsilon > 0$, $c < 1$

ASYMPTOTICS:

$f \in \Theta(g)$ ≈	$f \sim g$ =	$f \in O(g)$ ≤	$f \in o(g)$ <	$f \in \omega(g)$ >	$f \in \Omega(g)$ ≥
$f \in O(g)$ & $f \in \Omega(g)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$	$\exists c > 0\ \exists n_0 \geq 0$ $\forall n \geq n_0\ f(n) \leq cg(n)$ lim f/g = finite	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$	$g \in O(f)$
$2n + \log n \in \Theta(n)$; $n \in \Theta\left(\frac{3n^2}{(n+1)(n-1)}\right)$	$n+1 \sim n$; $n^2+n \sim n^2$	$\log n \in O(n)$; $n^{\log_3 2} \in O(n)$	$n^7 \in o(1.01^n)$	$\sqrt{n} \in \omega(\log^{300} n)$; $n^2 \in \omega(n)$	$f(n) = 1 + (n\sin(n\pi/2))^2$; $g(n) = 3n$

o → O
ω → Ω
O & Ω ↔ Θ

AMORTIZED COST (T):

if any sequence of m ops. (starting from empty d.s.) takes ≤ mT
potential func $\phi(A) \geq 0$
*decr. greatly when expensive op. called
for all ops. from $A_i \to A_{i+1}$ that take time t,
A.C. = $t + \phi(A_{i+1}) - \phi(A_i)$
Ex: $\phi(A) = c\left|n - \frac{m}{2}\right|$ c=const, n=# elts, m=capacity

WORD RAM: math model, step/ops
WORD: w bits. fit in 1 reg./mem if # < 2^w
LOAD: R[j] ← M[R[i]], STORE: M[R[i]] ← R[j]
- treat w = O(logn) for index all n input words
- $2^w \geq n \to w \geq \log n$. can address all n items in mem.
- arithmetic, logic, bitwise ops, mem all O(1)

INTERFACES:

STACK: LIFO
- push(x)
- pop()
- peek()
- isempty()

QUEUE: FIFO
- enque(x)
- deque(x)
- peek()
- len()

DEQUEUE: double-end queue
- push_front
- push_back
- pop_front
- pop_back
- front()
- back()
- len

SET: key-value pairs
- build
- find(k)
- unique keys
- multiset: can be non-unique (chaining)
- easy to find by key

SEQUENCE: maintain order; only have indices

Container	build(A)	given an iterable A, build sequence from items in A
	len()	return the number of stored items
Static	iter_seq()	return the stored items one-by-one in sequence order
	get_at(i)	return the i^{th} item (x_i)
	set_at(i, x)	replace the i^{th} item with x
	index_of(x)	returns the index i of item x
Dynamic	insert_at(i, x)	add x as the i^{th} item (shifts $x_i, \ldots$ to the right)
	delete_at(i)	remove and return the i^{th} item (shifts $x_{i+1}, \ldots$ to the left)
	insert_first(x)	add x as the first item
	delete_first()	remove and return the first item
	insert_last(x)	add x as the last item
	delete_last()	remove and return the last item

Container	build(A)	given an iterable A, build sequence from items in A
	len()	return the number of stored items
Static	find(k)	return the stored item with key k
Dynamic	insert(x)	add x to set (replace item with key x.key if one already exists)
	delete(k)	remove and return the item with key k
Order	iter_ord()	return the stored items one-by-one in key order
	find_min()	return the stored item with smallest key
	find_max()	return the stored item with largest key
	find_next(k)	return the stored item with smallest key larger than k
	find_prev(k)	return the stored item with largest key smaller than k

RECURRENCES: inductive sequence

$T(n) = aT(\frac{n}{b}) + f(n)$

PLUG & CHUG:

$T(n) = aT(\frac{n}{b}) + f(n)$
$T(\frac{n}{2}) = a\left(aT(\frac{n}{b^2}) + f(\frac{n}{b})\right) + f(n)$
$= a^k T(\frac{n}{b^k}) + \_n^? \sum_{i=\_}^{\ldots} \_^i$ ← get closed form
= … simplify
let $k = \log_b n$ and we know $b^k = n$
$T(n) = a^{\log_b n} T(1) + \ldots$
(if prove): let P(n) := T(n) = … (induction)

LOG PROPERTIES:

$\log_a b = \frac{\log_x b}{\log_x a}$
$\log(x^2) = 2\log x$
$\log(ab) = \log a + \log b$
$\log_a b = \frac{1}{\log_a b}$

DERIVATIVES:

$\frac{d}{dx}(\log_a x) = \frac{1}{x \ln a}$
$\frac{d}{dx}(\ln x) = \frac{1}{x}$
$\frac{d}{dx}(a^x) = a^x \ln a$

IMPLEMENTATIONS:

DOUBLY LINKED:
- val, prev, next
- can track tail

LINKED LIST:
- pointer-based
- val, next
- ins/del in Θ(1)
- isempty in Θ(1)

top
head
'a' 'b' 'c'

DYNAMIC ARRAY: expands/shrinks
- expands when doubling 2x
- when < n/4 elts, halve & move elts.
- avg. a series of expensive ops. over series of cheap ones
- good for get_at(i) → O(1)

1 2 3 4

HASHING: SUHA
- lookup by key
- pairwise indep.
- chain len O(α+1), $\alpha = \frac{n}{m}$ load factor
- n = total items inserted
- m = len of array
- L = bucket size
- collisions → chaining (points to DAA)

$P[h(k_i) = h(k_{new})] = \frac{1}{m}$ (suha)
$E(L) = 1 + P_r = 1 + \frac{n}{m} = 1 + \alpha \in O(1)$
LOAD FACT. α: avg. # items/slot (n/m)

A 0 1 2 3 4 5 … m-1
O(1) L

STABLE: preserve order: elts. w/ = keys
IN PLACE: no added memory

COMPARISON MODEL:
- binary comparisons
$\Omega(\log(n!)) \in \Omega(n\log n)$

SORTING:

INSERTION Θ(n²)
- comparing
SEAN
ESAN
AESN
AENS
S: ✓
IP: ✓

SELECTION: Θ(n²)
- handpicking
SEAN
EANS
EANS
AENS
AENS
S: ✗
IP: ✓

MERGE SORT: Θ(nlogn)
- stable: prefer left arr.
8 6 7 5 3 0 9 div. & conq.
"two finger"
6 7 8 0 3 5 9
0 3 5 6 7 8 9
S: ✓
I.P.: ✗
T(n) = Θ(n) + 2T(n/2)

DIVIDE & CONQUER: O(logn)
- base case: if n = _: …
- let m = ⌊n/2⌋
- if A[m] …, recurse on A[m:]
- else, recurse on A[:m+1]
- prove correctness on B.C., both sides, no overlap

RADIX SORT: Θ(cn), Θ((n+b)d) or $\Theta(n + n\log_n u)$
- must be bounded
- b = # buckets
- tuple sort w/ aux. counting sort
- if every key < n^c → linear O(cn)
- whole #s ← # digits = c
- if n words of len d in base b ← buckets
each round Θ(n+b), total time Θ((n+b)d)
S: ✓
IP: ✗
pseudopoly!!
- sort LSB → MSB
(720, 355, 436, 457, 657, 329, 839)
(720, 329, 436, 839, 355, 457, 657)
(329, 355, 436, 457, 657, 720, 839)

DIRECT-ACCESS ARRAY: Θ(u)
- all keys distinct
- u = possible range
S: ✓
I.P.: ✗
[0 _ 2 _ _ 5 _ _]
0 1 2 3 4 5 6 7

COUNTING SORT: Θ(n+k)
- keys don't have to be distinct - have a sequence within list
- use queue, etc. for stable
[_ _ _ _ _ _ _ _]
0 1 2 3 4 5 6 7
S: ✓
I.P.: ✗
k = bucket
n = len. arr.
['a', 'b'] pseudopoly!!

TUPLE SORT using ___:
- need auxiliary algo.
- $\Theta(n + n\log_n u)$ ← counting s.
- linear if $\log_n u \in O(1)$
[17, 3, 24, 22, 12] → [(3,2), (0,3), (4,4), (4,2), (2,2)]
5a + b
(0,3) (2,2) (3,2) (4,2) (4,4)

Algorithm	Time	In-place?	Stable?	Comments
Insertion Sort	$\Theta(n^2)$	✓	✓	$O(nk)$ for k-proximate
Selection Sort	$\Theta(n^2)$	✓	×	$O(n)$ writes
Merge Sort	$\Theta(n \log n)$	×	✓	optimal comparison
DAA Sort	$\Theta(u)$	×	N/A	u is domain of possible keys
Counting Sort	$\Theta(n+u)$	×	✓	linear if $u \in \Theta(n)$
Radix Sort	$\Theta(n + n\log_n(u))$	×	✓	linear if $u \in \Theta(n^c)$

BINARY TREE:

- depth: # edges from node → root
- height: longest # edges to leaf
- perfectly balanced: T full @ every level
h = O(logn)

O(1) left. rotate (u)
right. rotate (L)

- skew(n): h(n.right) − h(n.left)
N(h) = 1 + N(h−1) + N(h−2)
skew = 2: $n_h = n_{h-1} + n_{h-2} + 1$
$n_h > 2n_{h-2}$ (lower bound)
$n_h > 2(2n_{h-4})$
$n_h > 2^{h/2}$
$h < 3\log n_h < 3\log n \in O(\log n)$

- in-order: L, N, R → SQUIRTLE
- pre-order: N, L, R → UQSLRITE
- post-order: L, R, N → SQITRELU
O(n)

AVL PROPERTY: ∀n∈T, |skew(n)| ≤ 1 h ∈ O(logn) ✗ I.P.
- at most 2 rot. to fix violations find(x) = O(h)
- AVL sequence: maintains order of elts.
nodes contain: height/size of subtree
- AVL set: BST property (comparison model)
insert, del, search O(logn)

AUGMENTATIONS: dynamic order stats, computed for nodes from itself & left/right
build O(nlogn)

ORDER-STATISTIC TREE: os_select(T, i), os_rank(T, x)
- T.size = T.left.size + T.right.size + 1
- ith order: return element w/ ith smallest key
- rank: give element → return position i in ascending ord
indexes → i

INTERVAL TREES: interval_search(T[a, b])
- find node who's interval intersects i if exists
- T.low ≤ T.high
- keyed by max T.high
go left if T.left max ≥ a
- while node T doesn't overlap, traverse down tree.
- if left.max ≥ T.a → go left. if left.max < T.a, skip.

[16, 21] max = 30
[8, 9] max = 9
[25, 30] max = 30

del(3): O(log n)
- find inorder successor
- copy nodes val into del. node
- del. successor's node
- successor = leftmost child in x's right subtree

BST:
- distinct keys
- all keys on left subtree < node
- all keys on right subtree > node
- uses comparison model
- ins/find/del → w.c O(n), height n w.c.

STRONG IND:

we use str. ind. to show that P(n) = …
BASE CASE: n ≤ _. explain why.
IND. STEP: assume alg. works for all inputs ≤ n−1. WTS n works
typically pf. by cases

T(n) < 4T(n/2) + O(n)
a = 4, b = 2
$\log_b a = 2$
w = n
w = 2n
w = 4n $\frac{n}{4}$
#levels: $\log_2 n$
$f(n) = O(n) = O(n^{2-\varepsilon})$ → case 1
#nodes @ level i: 4^i
work per node @ level i: $O(\frac{n}{2^i})$
$\sum_{i=0}^{\log_2 n} c \cdot \frac{n}{2^i} 4^i = cn \sum_{i=0}^{\log_2 n} 2^i = cn 2^{\log_2 n} \sum_{i=0}^{\log_2 n} 2^{-i} = 2cn \cdot n \to O(n^2)$

Problem 2. Nesting

Klefki wants to store a SET of locks. Each lock x has two keys: a comparable key $k_1(x)$ and a hashable key $k_2(x)$. The ordered pair $(k_1(x), k_2(x))$ is a unique identifier, but neither key need individually be unique. Help Klefki implement a SET data structure in which the FIND and DELETE operations can take keys from *either or both* key spaces. (If FIND is given only one type of key, it can return any lock with that key. If DELETE is given only one type of key, it should delete every lock with the given key.) All three operations should run in $O(\log n)$ time. Analyze the runtime of each, including classifying each as worst-case, amortized, and/or expected. Prove tighter bounds when possible. COMPARABLE / HASHABLE
- each lock lives in A & H

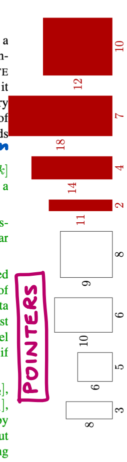
Solution: Keep two nested data structures. The first is an AVL set A, keyed by k_1. Elements $A[k]$ are hash sets, keyed by k_2, that store all the locks x with $k_1(x) = k$. The second is inside-out: a hash set H (keyed by k_2) of AVL sets (keyed by k_1).

INSERT(x) adds x to both $A[k_1(x)]$ and $H[k_2(x)]$ (possibly after initializing empty sets if necessary). This uses both AVL and hash insertion operations, both on data structures of up to linear size. Runtime is hence expected and amortized $O(\log n)$.

DELETE(k) if given two keys deletes $A[k_1][k_2]$ and $H[k_2][k_1]$, which takes expected and amortized $O(\log n)$ time. If given only one, it either clears $A[k_1]$ or $H[k_2]$, depending on which kind of key is given. Each deleted lock then gives the second key necessary to delete from the other data structure. Each lock deleted takes expected and amortized $O(\log n)$ time. We amortize this cost over prior insertions (each deleted lock was inserted at least once). We are left with the top-level FINDs, which cost $O(\log n)$ expected amortized time if given two keys, $O(\log n)$ amortized if given only k_1, or $O(1)$ expected amortized if given only k_2.

FIND if given two keys simply looks for $A[k_1][k_2]$. If given only k_2, it returns the root of $H[k_2]$, which takes expected $O(1)$ time. If given only k_1, it must return a representative from $A[k_1]$, which (if maintained) takes $O(\log n)$ worst-case time. We can maintain this representative by cross-linking each hash set in A with a linked list that maintains insertion order. Note: without maintaining the representative, we can find an element in a hash table by probing randomly, giving $O(\log n)$ expected time.

POINTERS



Solution:
1. If b is red, then b.opp_larger = 0; if b is white, then b.opp_smaller = 0.
2. Initialize a counter w = 0 to track how many white boxes have been processed.
3. While either list is non-empty, we check red[0].width > white[0].width (if either list becomes empty, pretend for the sake of comparison that it contains a dummy box of infinite width).
   - If so, set white[0].opp_larger = len(red), then remove white[0] and increment w.
   - Otherwise, set red[0].opp_smaller = w, then remove red[0].

Proof of correctness. All red boxes have larger length than all white boxes by assumption, so the first line is correct. All boxes are processed in increasing order of width. Therefore, when a red box is processed, all white boxes that have already been processed have smaller width and length, and all white boxes that have yet to …

(b) Describe (with proof) an augmentation from which CLOSEST() can be computed in constant time.

Hint: Use an ordered triple that captures the difference between the examples you described above. closest = min + diff. b/t 2 elts.

Solution: In the given solution, the *minimum* value in the right subtree changed without changing any augmentation values. Generally, the closest pair can be in the left subtree, be in the right subtree, or contain the root value. In order to identify the last case, we want to know the minimum and maximum values of each subtree. The triple (min, CLOSEST, max) is a valid augmentation that can be maintained via:

CLOSEST, AUGMENTATION

$$\min(T) = \min(\min(T.\text{LEFT}), T.\text{ITEM})$$
$$\max(T) = \max(\max(T.\text{RIGHT}), T.\text{ITEM})$$
$$\text{CLOSEST}(T) = \min \begin{cases} \text{CLOSEST}(T.\text{LEFT}) \\ \text{CLOSEST}(T.\text{RIGHT}) \\ |T.\text{ITEM} - \max(T.\text{LEFT})| \\ |T.\text{ITEM} - \min(T.\text{RIGHT})| \end{cases}$$

Problem 2. [10 points] **Pidgey** (1 part)

Help Pidgey design an algorithm that takes as input an array A of n integers, and outputs a pair of indices $i \neq j$ (if they exist) such that $A[i] - A[j]$ is a multiple of $\lceil \log n \rceil$. If there are many such pairs, you may output any one of them. Briefly justify correctness. Analyze the runtime, including proving matching uppe and lower asymptotic bounds. Classify the runtime as worst-case, expected, and/or amortized.

Choose your own adventure: PIGEONHOLE + DAA

- ■ For full credit, your algorithm must run in $O(\log n)$ time.
- ☐ For up to 5 points, your algorithm must run in $O(n)$ time.

Solution: Create a DAA that stores i at location $\text{rem}(A[i], \lceil \log n \rceil)$. Output the colliding indices from th first collision. $A[i]$ to i using the hash function $h(k) = \text{rem}(k, \lceil \log n \rceil)$. Output the colliding indices fror the first collision.

Correctness follows from the definition of modular equivalence and the fact that $\text{rem}(a, c) = \text{rem}(b, c)$ if $a \equiv b \mod c$.

Every insertion takes worst-case $O(1)$ time because there is at most one collision by construction. B pigeonhole principle, the first collision is found within the first $\lceil \log n \rceil + 1$ elements, so runtime is worst case $O(\log n)$. This bound is tight, witnessed by the input array $[1, 2, \ldots n]$, for which the collision betwee 1 and $\lceil \log n \rceil + 1$ is found after $\log n$ insertions.

(b) [16 points] Suppose instead that Riolu knows that there exists some $\sigma < n^{1210}$ such that her friend can be partitioned into n σ-pairs with one left over. Design a worst-case $O(n)$ algorithm to find th unpaired Pokémon's aura, given as input an array A of the $2n+1$ Pokémon's auras. Note that σ is unknown. Prove your algorithm correct, and analyze its runtime. You may assume a correct solution to part (a), even if you did not solve it yourself.

Choose your own adventure: RADIX SORT

- ■ For full credit, your algorithm must output the unpaired aura.
- ☐ For up to 8 points, your algorithm need only decide whether or not the largest aura is unpaired.

Solution: We reduce to part (a).

1. If any aura is larger than n^{1210}, return it.
2. Radix sort A.
3. Check if for every index $i \in [1, n]$, $A[i] + A[-i]$ is the same (and $< n^{1210}$); if so, return $A[0]$.
4. Repeat the previous step on the reverse of A.
5. Return the result of part (a), with $\sigma = A[0] + A[-1]$.

Proof of correctness: Auras are positive, so every paired aura must be less than $\sigma < n^{1210}$. Hence step 1 is correct or a no-op. We now condition on whether the unpaired aura is the smallest, largest, or neither.

- The smallest aura is unpaired. In this case, because A is sorted, $A[i]$ and $A[-i]$ must form a σ-pair for every i. This is detected by step 3.
- The largest aura is unpaired. This case is symmetric to the previous and is detected by step 4.
- The smallest and largest auras are both paired. In this case, they must be paired with each other, so we know σ and can reduce to part (a).

Runtime analysis: Steps $1, 3, 4$ are linear scans. If step 1 does not return, then auras are polynomial, and step 2 takes linear time. Step 5 takes linear time by assumption. Total runtime is linear.

Problem 4. [20 points] **Happy Go Lucky** (1 part)

Blissey's Pokémon friends are at integer locations along a line. Blissey is keeping them happy by placing Lucky Eggs. Each Lucky Egg e is represented as a pair of integers (a, b) with $a \leq b$, and it makes all Pokémon at locations x with $a \leq x \leq b$ happy. Blissey's friends may accidentally break the Lucky Eggs, which then lose their effect.

Blissey would like to know who is happy. Design a data structure that supports the following operations:

AVL BST, AUGM.

- PLACE(e): Add a Lucky Egg e to the data structure
- BREAK(e): Delete a Lucky Egg e from the data structure
- HAPPY(x): Output TRUE iff the Pokémon at location x is currently happy

Your data structure must use $O(n)$ space, and all operations should have worst-case $O(\log n)$ runtime, where n is the number of Lucky Eggs currently in existence. You need not prove runtime or correctness.

Choose your own adventure:

- ■ For full credit, implement the operations as given above.
- ☐ For up to 10 points, assume that there exists d such that all Lucky Eggs have the form $(a, a+d)$. Your runtime and space requirements must not depend on d.

Solution: Keep all Lucky Eggs in an AVL BST keyed lexicographically by either (a, b) or $(a, -b)$: that is, they are primarily keyed by a, and if multiple eggs have the same a, ties are broken by some ordering of b. Augment each node with the maximum b in each subtree, computed by $node.maxb = \max\{node.item.b, node.left.maxb, node.right.maxb\}$. PLACE and BREAK are standard add and delete operations.

HAPPY(x): Recursively search from the root. At each *node*, if *node* is null, output FALSE. Otherwise:

1. If $node.item.a > x$, recurse on $node.left$ and output its value. (The only eggs that can possibly have $a \leq x$ are all in the left subtree.)
2. Else, $node.item.a \leq x$:
   (a) If $node.item.b \geq x$, output TRUE.
   (b) If $node.left.maxb \geq x$, output TRUE. (All eggs in the left subtree have $a \leq x$, and this condition means at least one of them have $b \geq x$.)
   (c) Otherwise, recurse on $node.right$ and output its value. (The only eggs that can possibly have $b \geq x$ are all in the right subtree.)

(b) Suppose instead that every Pokémon will train for more than one hour, i.e. $b_i - a_i > 1$. (You may no longer assume that a_i is an integer.) Give an $O(n)$ time algorithm to solve the problem. You may assume a correct solution to part (a), even if you didn't manage to solve it yourself. I_1 = [1.2, 4.5] → c_1 = 2 → [1.2, 2], [2, 4.5]

Solution: Reduction to (a) use counting sort on right & left respectfully, then merge unions

1. For every i, there is an integer c_i such that $a_i < c_i < b_i$.
2. Compute $\bigcup\{[n - c_i, n - a_i]\}$ using part (a).
3. Replace each interval $[x, y]$ in the union resulting from step 2 with $[n - y, n - x]$.
4. Compute $\bigcup\{[c_i, b_i]\}$ using part (a).
5. Merge the two lists into a single sorted list S.
6. Iterate over S, merging each interval with the previous interval if they overlap.
7. Return S.

Proof of Correctness: We are computing the function $f(\mathcal{I}) = \mathcal{J}$, where $\mathcal{I}$ and $\mathcal{J}$ are finite sets of intervals with the same union, and $\mathcal{J}$ is minimal. Because c_i is defined as an integer, we know that $n - c_i$ is also an integer, and step 2 correctly finds the union of $[n - c_i, n - a_i]$ from the correctness proof of part (a).

To show that steps 2 and 3 correctly output the union of $[a_i, c_i]$, we observe that both steps apply the transformation $g(\mathcal{I}) = \{\{n - x : x \in I\} : I \in \mathcal{I}\}$, which is self-inverting and commutes with f. (1)

Step 4 is correct because c_i is an integer and thus correctness from part (a) applies. The remaining steps are correct per the proof from (a).

Aside: This level of detail is not needed, but for a full proof of (1), it suffices to show that $g(f(g(\{[p,q],[r,s]\}))) = f(\{[p,q],[r,s]\})$, for any p, q, r, s, as this can then be applied inductively.

WLOG, assume that $p < r$. Then, we have two cases:

- If $q < r$ then the two intervals are disjoint. Similarly, $n - r < n - q$ and $n - r < n - p$, so $[n - q, n - p], [n - s, n - r]$ are also disjoint. On disjoint

- `find_best(m)`: Find the highest rated player whose cost is between $m/2$ and m (inclusive), or report that no such player exists

All of `insert`, `delete`, and `find_best` must still have $O(\log n)$ runtime.

Hint: There are two very similar solutions to part (a), but one of them can be adapted much more naturally to solve part (b) as well. * key by cost

Solution: The first solution can be adapted; the second should not. We keep three augmentations: the best player in each subtree, the min key in each subtree, and the max key in each subtree Both of `insert` and `delete` are unchanged. We adapt `find_best`: AUGMENTATION/RANGE QUERY

1. If the tree is empty, the min key is greater than m, or the max key is less than $m/2$, then output $\perp$
2. If the min and max keys are both in the interval $[m/2, m]$, then output the best augmentation
3. If the root cost is greater than m, recurse on the left
4. If the root cost is less than $m/2$, recurse on the right
5. Otherwise, recurse on both children, and output whichever of these two calls or the root player has the best rating.

Correctness: Step 1 outputs $\perp$ iff there are no costs in the range $[m/2, m]$. Step 2 executes iff all costs are in the range $[m/2, m]$, in which case they are all eligible. In this case, the best augmentation is the correct output by definition. Steps 3 and 4 execute iff the root is not eligible, in which case one subtree is also ineligible, and recursion on the other subtree gives the correct output. Step 5 is a catch-all and is correct by casework on where the correct output is.

Runtime: The steps are mutually exclusive. Steps 1-2 return immediately, and Steps 3-4 make a single recursive call on a subtree. By the guards on the previous steps, Step 5 executes iff the root cost and at most one extremum are in the range $[m/2, m]$. This means at least one of $m/2$ or m is strictly between the min and max keys. Suppose Step 5 executes on two subtrees T_1 and T_2, neither of which is an ancestor of the other. By the BST property, their least common ancestor has a key between them, so WLOG we have $\min T_1 < m/2 \leq T_1.\text{root} \leq \max T_1 \leq \min T_2 \leq T_2.\text{root} \leq m < \max T_2$. Therefore the right child of T_1 and the left child of T_2 both return immediately. This means there are at most four recursive calls at every level of the recursion tree, giving $O(\log n)$ runtime.

Problem 3. Long Jump 2 AVL TREES

Henry is unsatisfied with the current long jump rules and wants to give athletes prizes according to his own set of rules. He arrives at the competition with a bag full of Snickers bars to give the athletes. Whenever he's feeling generous, he selects a minimum distance threshold t. An athlete is eligible to receive this prize if their most recent jump was at least distance t. He gives the prize to the eligible athlete who jumped most recently, or to Srini if no athlete is eligible. Athletes have no limits to their number of attempts.

Henry needs a data structure that can support the following operations:

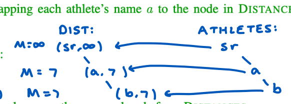
- JUMP(a, d): Record that an athlete with the name a just achieved a long jump distance of d
- PRIZE(t): Output the name of the prize winner, given distance threshold t

Describe a data structure that uses $O(n)$ space and implements both of the above operations in $O(\log n)$ time, where n is the total number of athletes who have participated so far. Briefly justify correctness and analyze runtime. You do not need to prove space complexity or runtime lower bounds. You need not analyze data structures or algorithms presented in class, but you must describe and analyze any modifications that you make. Assume that the JUMP operation is executed at the time of the recorded jump, but Henry cannot tell what this time is.

Solution: We store RECORDS as tuples (a, d), representing that the athlete with name a achieved a jump of distance d.

We keep two AVL trees:

- In DISTANCES, we store a SEQUENCE of $n+1$ RECORDS, in order of time. Initially, the only RECORD is (Srini, ∞). We add the augmentation MAX(T), which is the maximum RECORD in T's subtree, comparing by distance. We compute this by comparing the root RECORD's size with the augmentations of both children.
- In ATHLETES, we store a set mapping each athlete's name a to the node in DISTANCES containing a record (a, d).



Operations are implemented as follows:

- JUMP(a, d):
  - Remove a from ATHLETES, and remove the removed node from DISTANCES
  - Add (a, d) to the end of DISTANCES, and in ATHLETES map a to the new node
- PRIZE(t): (defined recursively and wrapped, starting with DISTANCES)
  - If T's right child augmentation is at least t, recurse on right
  - Else if T's root RECORD has distance at least t, output it
  - Else recurse on left

Runtime analysis. We store two AVL trees of size $n+1$, for a total of $O(n)$ space. Augmentation runtime is immediate from definition. JUMPing performs at most four AVL insertions / deletions, each of which takes $O(\log n)$ time. PRIZE follows at most a path from the root of DISTANCES to a leaf, which takes $O(\log n)$ time.

☐

Proof of correctness. DISTANCES stores every Athlete's most recently jump attempt, plus a dummy for Srini. We observe that WLOG these are the *only* jumps that have been attempted, as no others affect prize distribution.

Correctness of MAX augmentation is immediate from definition.

When JUMP is called, the new RECORD is necessarily the most recent. It should therefore be added at the end of DISTANCES and replace any RECORD from the same Athlete.

Correctness of PRIZE follows from induction and the traversal order of DISTANCES, noting that Srini is always "eligible". Assume that T contains an eligible Athlete and that PRIZE executes correctly on smaller trees. We condition on where the prize winner is.

- If the athlete is in the right subtree, then they are also the last eligible Athlete in the right subtree, so are found by the first recursive call.
- If they are at the root, then all Athletes later in traversal order, i.e. in the right subtree, are ineligible. The maximum size appearing in the right subtree is less than t, so the first recursive call isn't executed, and the root is returned.
- If they are in the left subtree, then they are the last eligible record in the left subtree. All Athletes later in traversal order, i.e. the root and everyone in the right subtree, are ineligible. The else clause is executed, and recursion on the left finds the correct prize winner.

Problem 1. Counting Sheep

Mareep is implementing a counter that stores a natural number k (initially 0) in base-n using an array A of length n. The counter has two operations: base n

- `increment()` : increments k by one
- `get()` : outputs and resets k

DYNAMIC ARR, POT. FUNC.

Describe how to implement both operations, and prove that they take amortized constant time.

Solution: For `increment`, we start by incrementing $A[0]$. Each time $A[i]$ overflows, we reset it and increment $A[i+1]$. If $A[n-1]$ overflows, double n, reallocate A, and set $A[n/2] = 1$.

For `get`, output A, reset $n = 2$, and reallocate A.

To analyze runtime, we want to figure out a good potential function. The first observation is that `increment` becomes expensive when $n - 1$ appears many times in A, but it also gets rid of those appearances. Therefore these values should contribute to the potential. The second observation is that `get` must always use linear work, but this work cannot exceed the number of earlier `increments`. We define $\phi(A) = c(m + k)$, where c is a constant and m is the number of indices i such that $A[i] = n - 1$. (Note that $\phi(A) = c(2m + n)$ will also work.)

The work done for `increment` is $O(m' + 1)$, where m' is the length of the longest prefix of A whose values are all $n - 1$. The potential increases by $O(1)$ and decreases by $\Omega(m')$. By choosing c to be sufficiently large, the potential decrease can be made to dominate the work done, so the amortized cost is $O(1)$.

The work done for `get` is $\Theta(n)$, and the potential decrease is $2^{\Omega(n)}$, so there is no amortized cost.

(b) Assuming unique node values, give a worst-case $\Theta(n^2)$ algorithm to determine Bulbasaur's `PreOrder` traversal given Ivysaur's `InOrder` and Venusaur's `PostOrder` traversals of a tree T. Prove your algorithm correct (and observe that it proves that the `PreOrder` traversal is *uniquely* determined by the `InOrder` and `PostOrder` traversals). Analyze the runtime of your algorithm, including both upper and lower bounds. PREORDER ← POST. IN

Solution: We know that the root is always the last item in the `PostOrder` traversal. Next, we know that the `InOrder` traversal has the structure `Left_SubTree, Root, Right_SubTree`, so we can search the `InOrder` for the root. We know that the left subtree contains all nodes to the left of the root in the `InOrder` traversal and the right subtree contains all nodes to its right. sImilarly, all left nodes appear before any of the right nodes in the `PostOrder`. So, once we find the index of the root in the `InOrder` traversal, we can also split at this index in the `PostOrder` to have that split into left and right subtrees.

Once we have the traversals split into the two subtrees, we recurse and run the same procedure again, first with the left subtree, and then with the right subtree.

We then form the `PreOrder` traversal by placing first the root, then the `PreOrder` of the left subtree and finally the `PreOrder` of the right subtree.

Correctness: We are correctly obtaining the root node in each recursive call as the root node will always be the last item in the `PostOrder` traversal. We are also splitting at the correct node to determine left and right subtrees since each node value is unique.

Runtime: The worst case runtime is $O(n^2)$ since each node will be the root exactly once, and each time a node is the root we have to search for it in the `InOrder`, which takes at most $O(n)$ time (the length of the list of nodes we have to search through). We are able to show this bound is tight by considering the case where the `InOrder` and `PostOrder` are the same, making a tree that is a chain of left children. This exhibits worst case behavior since each time we have to search for the root we must iterate through all of the remaining nodes in the traversal, giving a runtime of $\sum_{i=0}^{n-1}(n - i) = \Theta(n^2)$.

Problem 2. Rocket Recruits BS, SORTING, PAIRS

Giovanni has recruited n new Grunts for Team Rocket. For each $i \in \{0, \ldots, n-1\}$, Grunt i has k_i Pokémon. All of the k_i are positive and distinct.

Giovanni wants to pair them into teams with exactly p Pokémon between them. A pair of Grunts (x, y), with $x < y$, such that $k_x + k_y = p$ is called a *battle pair*.

Giovanni wants to find two quantities: the number B of battle pairs he can make, and the battle pair (x^*, y^*) for which x^* is minimal.[1]

(a) Describe an algorithm to find B and (x^*, y^*) with a worst-case runtime of $O(n \log n)$. Prove your algorithm correct, and analyze its runtime.

Solution: We present two solutions. The first is simpler, and uses binary search to find a valid y that pairs with each x after sorting the list of Pokémon quantities. The second solution improves on the binary search step, replacing it with a two-finger algorithm to find all battle pairs in linear time. While the second algorithm is more efficient in practice, both have the same asymptotic runtime $\Theta(n \log n)$ due to the sorting step being the bottleneck.

For all parts of the question, we denote "*record* (x, y) as a battle pair" as incrementing the running count B, and then updating the running optimum $(x^*, y^*) \leftarrow (x, y)$ if $x < x^*$.

ci = [(4,0), (7,1), (1,2), (9,3)] (elt, ind)

Algorithm 1 (Sort + Binary Search): Sort: [(1,2), (4,0), (7,1), (9,3)] b.s. for ky = p − kx

1. Initialize the running count $B = 0$ and running optimum $(x^*, y^*) = (n, n)$.
2. Keep track of the original indices of each element in $\{k_i\}$ before sorting. This can be done by creating a new sequence $\{c_i\}$, and setting $c_i = (k_i, i)$ for each $0 \leq i \leq n - 1$.
3. Sort $\{c_i\}$ by their Pokémon quantities (the first element k_i of each tuple), using a $\Theta(n \log n)$ sorting algorithm such as merge sort.
4. For each $x \in [0, n-1]$, binary search on $\{c_i\}$ to find a tuple (k_y, y) with $k_y = p - k_x$. If there exists such a y, and $x < y$, record (x, y) as a battle pair.
5. Return S and (x^*, y^*).

Proof of correctness. It is easy to see that if we record exactly the set of all battle pairs once each, the correctness of S and (x^*, y^*) follows naturally. We show that this is indeed what we record in Step 4.

Any battle pair (x, y) must satisfy $k_y = p - k_x$, so when iterating through x, the binary search will correctly find the unique y with Pokémon quantity $p - k_x$, and then record (x, y). Likewise, we can show that any pair that's not a battle pair will not be found and recorded. Thus, we record all and only each battle pair exactly once.

Problem 4. Find the MissingNo.

Given an array A of strictly increasing integers of length n, and a number s, your task is to design an algorithm to find the smallest integer larger than or equal to s not in A.

(a) Design an algorithm to find the missing integer in time $O(\log n)$. You must describe your algorithm in English.

Solution: 1 (Modification of binary search): We modify the comparison step. When A is not empty, we find the index of the middle element i and check if $A[i] < s$. If so, we want to recurse on the right side of the list. If not, we check if there are consecutive numbers from s to $A[i]$ by finding the difference between $A[i]$ and s. Let's call this k. Next, we check if $A[i - k] == s$. If so, we know that there are consecutive numbers from s to $A[i]$, so we recurse on the right side of the list, setting s to $A[i] + 1$. If not, we recurse on the left side of A.

(b) Prove your algorithm correct.

Solution: 1: We proceed by strong induction on n. Our hypothesis is that our solution finds the smallest integer larger than or equal to s not in A correctly. The base case is when $n = 0$. In this case, we have an empty array and therefore, we return s, which is the smallest integer larger than or equal to s not in A. For the inductive step, let's assume that our hypothesis holds for all arrays of size $len(A) < n$ and let's show that for arrays of size n the algorithm is still correct. There are three cases.

1. The middle element is less than s: Since we have a strictly increasing array, any missing element in the list on the left side is smaller than s, which means that we can disregard the left side. According to our algorithm, we'll recurse on the right side of the array, and as the right side of the array will have size less than n, our inductive hypothesis ensures that the algorithm will run correctly on this side.
2. The middle element is larger than or equal to s and there is a consecutive sequence of integers in the list from s to the middle element: The number we're looking for cannot be in the left side as we cannot have any gaps in our strictly increasing array. Therefore, we need to look for a missing element in the right side of the array. Our algorithm does precisely that, and since the right side of the array has size less than n, the algorithm will find the correct output based on our inductive hypothesis.
3. The middle element is larger than s but there is not a consecutive sequence from s to the middle element: This means that there is a missing number on the left side of the list that is larger than s. Thus, we should search in the left side, which is what our algorithm does. Since the left side of the array has size less than n, our algorothm will reach the correct output based on our inductive hypothesis.

5. We saw how to create an iterator for an AVL tree in which the NEXT operation takes amortized constant time but worst-case logarithmic time. Describe how to change the AVL tree such that this NEXT operation takes worst-case constant time. The functionality of the AVL tree (including asymptotic runtimes) should otherwise be unchanged. CROSS-LINK: AVL+LL

Solution: Each node should additionally carry a pointer to its predecessor and successor. These can then be treated as doubly-linked list nodes as well as binary tree nodes. When we delete a node from the binary tree, we can also have it delete itself from the linked list in the usual way. For insertion, we observe that we can only ever add a leaf node, whose parent is either its successor or predecessor. The parent node can then add into the linked list in the usual way. This is called *cross-linking* the AVL tree with a linked list.

Alternatively, this will work with only a successor pointer and no predecessor pointer. We use the binary tree FINDPREV operation and then update both nodes' successor pointers. This takes $O(\log n)$ time, which is still dominated by the insertion or deletion runtime.