Surname:

First Name:

Matriculation No.:

Sample Solution

Karlsruhe Institute of Technology
Institute for Theoretical Informatics

Prof. Dr. Marvin Künnemann

12.03.2026

Exam Algorithms II

Problem 1.	Miscellaneous Tasks	12 Points
Problem 2.	Max-Flow: You shall not pass (more than once)!	10 Points
Problem 3.	Shortest Paths: Pairwise Shortest Paths	9 Points
Problem 4.	Randomized: Sibling Sabotage	9 Points
Problem 5.	Geometric Algorithms: Asteroids	9 Points
Problem 6.	Stringology: Evenly Spaced Patterns	11 Points

Please note:

- The only allowed aid is **one** A4 sheet with your **handwritten** notes.
- Write your answers in blue or black ink only, using an indelible pen.
- **Write** your **Matriculation No.** on **all** sheets of the exam and additional pages.
- The duration of the exam is **120 minutes**.
- You may write on the backside of the sheets. If you do so, please reference the exercise that you answer.
- The exam contains **18 pages**.

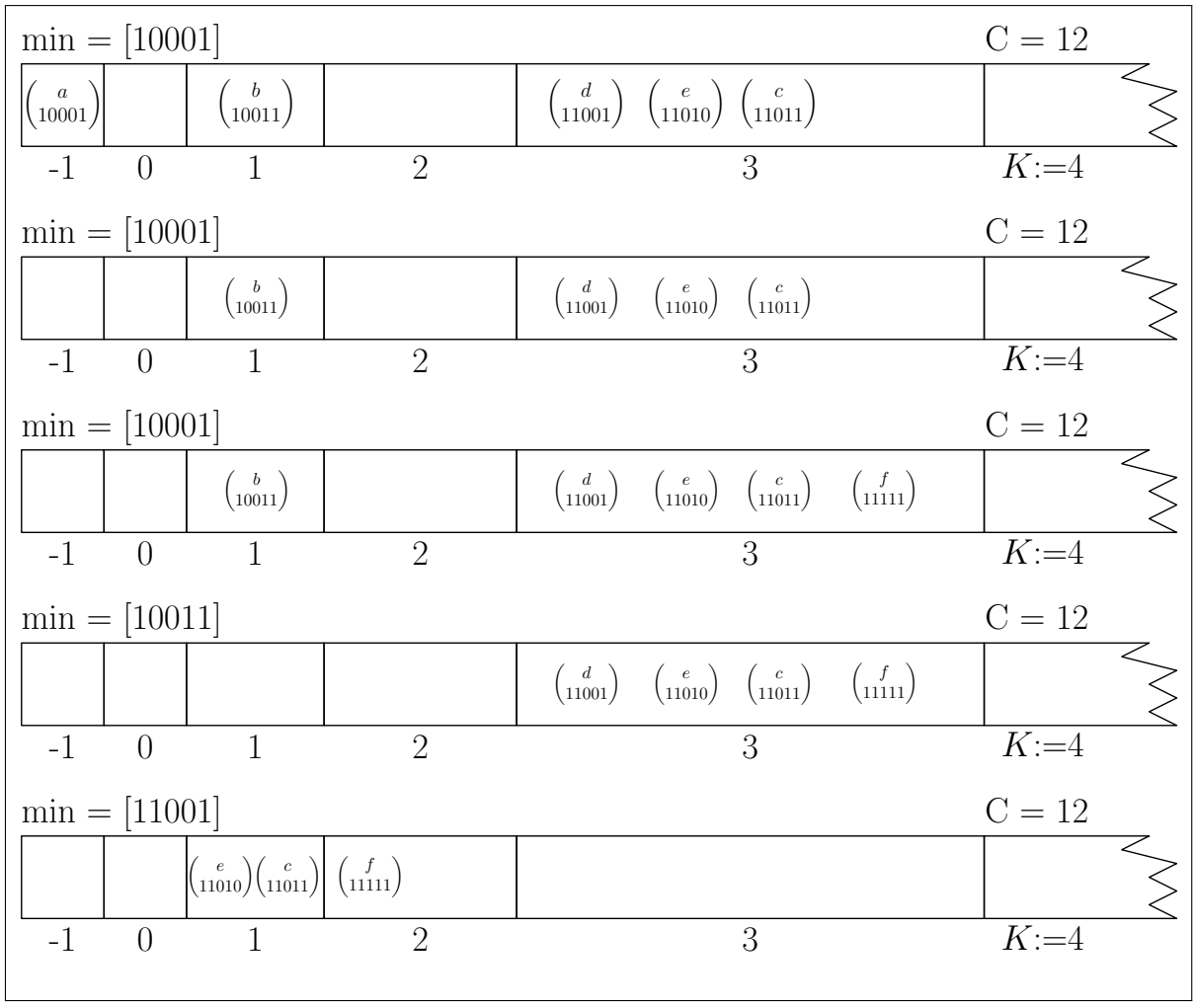
Sample Solution

Problem 1. Miscellaneous Tasks **(_ /12 P.)**

a. Given the Radix Heap below with $C = 12$ and $K = 4$, perform the following operations on it. Draw the Radix Heap after every operation. Do not forget to state the value of min (denoted d^* in the lecture). **(_ /4 P.)**
Note: It suffices to give the keys (a-f) without the numerical values when filling in the buckets.

- `deleteMin()`
- `insert(f,11111)`
- `deleteMin()`
- `deleteMin()`

Solution



b. Consider the following running times for input size n and parameter $k \in \mathbb{N}$. State whether they are fixed-parameter tractable and justify your answer. **(_ /2 P.)**

1. $\left(2^{k \log k}\right)^{\log^2 n}$
2. $k^k \cdot n^{\frac{1}{\sqrt{k}}}$

Solution

1. $\left(2^{k \log k}\right)^{\log^2 n} \geq n^{k \log k}$, as $n^{k \log k}$ depends on k, this running time is not FPT.
2. $k^k \cdot n^{\frac{1}{\sqrt{k}}} \leq k^k \cdot n$, as $\frac{1}{\sqrt{k}} \leq 1$ for all $k \in \mathbb{N}$, thus the runtime is in $\mathcal{O}(f(k) \cdot p(n))$, where $p(n)$ does not depend on k and therefore FPT.

c. Consider approximation algorithms with the following running times for input size n and approximation factor $(1+\varepsilon)$ for some minimization problems. State whether they are a PTAS, FPTAS or neither and justify your answer. **(_ /2 P.)**

1. $n^{\frac{1}{\varepsilon}} \cdot \log \frac{1}{\varepsilon}$
2. $\left(n + \frac{1}{\varepsilon}\right)^2 \cdot \log(\frac{n}{\varepsilon})$

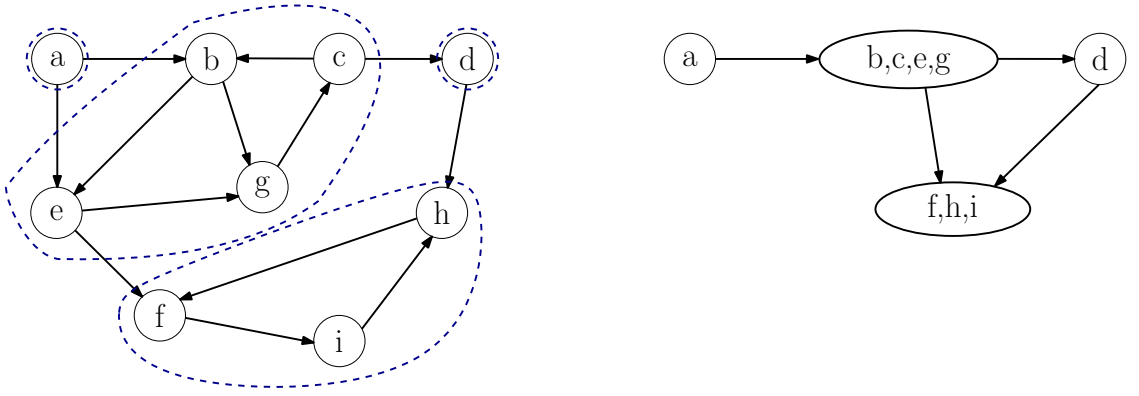
Solution

1. The running time is polynomial in n but not in $1/\varepsilon$ and thus the algorithm is a PTAS.
2. The running time is polynomial in both n and $1/\varepsilon$ and thus the algorithm is a FPTAS.

d. Mark all strongly connected components in the given graph and draw the shrunken graph. **(_ /2 P.)**

Two copies are provided below. Clearly mark the one that should be graded if you use both copies. Otherwise, we grade this part with 0 points.

Solution



Sample Solution

e. Consider an operator $\star : \mathbb{N}^2 \to \mathbb{N}$ defined as $x \star y := x + y + xy$. **(_ /2 P.)**
Let $a_1, \ldots, a_n$ be integers. Show that one can compute $(\ldots((a_1 \star a_2) \star a_3) \star \cdots \star a_n)$ in time $\mathcal{O}(n/p + \log p)$ using p parallel processors.

Solution

We first show that $\star$ is associative:

$$\begin{aligned}(a \star b) \star c &= (a + b + ab) + c + (a + b + ab)c \\ &= a + b + c + ab + bc + ac + abc \\ &= a + (b + c + bc) + a(b + c + bc) = a \star (b \star c)\end{aligned}$$

The operation can be performed in constant time, as it's two additions and a multiplication. Using the reduce-scheme known from the lecture, we thus obtain an $\mathcal{O}(n/p + \log p)$-time algorithm on p parallel processors.

Sample Solution

Problem 2. Max-Flow: You shall not pass (more than once)! **(_ /10 P.)**

In a game show called "You shall not pass (more than once)!", k participants are placed in an $n \times n$ grid-like maze of rooms connected by doors. To win, the participants must reach an *exit*, defined as any room located on the boundary of the maze. The maze features a unique restriction: each *door* can only be passed through **once** in total. Once any participant traverses a door, it is locked and becomes impassable for everyone else.
Formally, we represent the maze as an $n \times n$ undirected (incomplete) grid graph $G = (V, E)$, where V is the set of rooms and E is the set of available doors.

Rooms (V): Each room is identified by its coordinates (x, y), s.t. $V = \{v_{(x,y)} \mid 1 \leq x, y \leq n\}$

Doors (E): An edge $\{v_{(x_1,y_1)}, v_{(x_2,y_2)}\}$ exists if the corresponding door is open.

The rooms have to be adjacent in the grid i.e., $|x_1 - x_2| + |y_1 - y_2| = 1$.

Exits (B): The exit boundary $B \subseteq V$ consists of all rooms on the edges of the grid:

$$B = \{v_{(x,y)} \in V \mid x \in \{1, n\} \text{ or } y \in \{1, n\}\}$$

Starting Positions: There are k participants, each starting in a distinct room $s_1, \ldots, s_k \in V$.

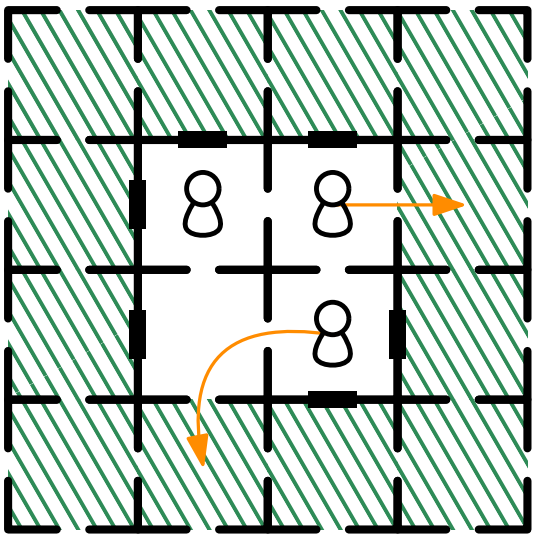
As an avid fan of the show, you are interested in determining for any instance, given as the $n \times n$ maze $G = (V, E)$ and the k starting positions $s_1, \ldots, s_k$, the maximum number of participants that can reach the exit at the same time.

Note: in the following, $T_{maxflow}(n, m)$ denotes the optimal time needed to compute the maximum flow in a network with n vertices and m edges.

a. Determine for the maze below with 3 participants how many can escape at the same time. It suffices to draw the paths that the participants can take to reach the exit. The black rectangles denote closed doors; rooms on the boundary are shaded. **(_ /1 P.)**

Solution

At most 2 participants can escape at the same time, e.g.



b. While you really like puzzling out how many participants can escape, your inner computer scientist wants to solve this problem algorithmically. **(_ /5 P.)**
Design an $\mathcal{O}\big(T_{\text{maxflow}}(n^2,m)\big)$-time algorithm that determines the maximum number of participants that can reach the exit for a given instance as described above. Prove the correctness (specifically, why your formulation as a max-flow problem models the problem correctly) and analyze the runtime of your algorithm.

Solution

We consider the graph representation of the maze. We transform this into a network by including a dedicated source and sink vertex denoted by s,t respectively. The network consists of all edges present in the maze plus edges from and to the source and sink. In particular, we construct the network such that every starting vertex s_i is connected to the source with capacity 1 and every vertex on the boundary B is connected to the sink with high capacity (i.e. at most k). Every edge has unit capacity.
Formally we define the network as $G' = (V \cup \{s,t\}, E', c)$ with $E^* = \{(u,v),(v,u) \mid \{u,v\} \in E\}$ where

$$E' = E^* \cup \{(s,s_i) \mid i \in [k]\} \cup \{(b,t) \mid b \in B\}$$

and for $b \in B$, let $c((b,t)) = k$ for all $(b,t) \in E$. The remaining edges all have unit capacities. We now compute the maximum flow in G' and return this value.

Correctness: We want to show that the value of a flow in the constructed network is exactly the number of (edge-)disjoint paths that can reach the exit from the starting positions.
Assume j participants can escape. Then there exist j edge-disjoint paths from $s_{i_1},\dots,s_{i_j}$ to the boundary vertices B. As these paths can be extended to s and t, their union forms a valid s-t-flow of value j.
Conversely, consider a flow of value j. By construction this flow passes through exactly j starting vertices $s_{i_1},\dots,s_{i_j}$ and boundary vertices $b_1,\dots,b_j$. As the network has only unit capacities (except edges to t), we can decompose the flow into j disjoint paths that each go through some s_i and b_i. These paths correspond exactly to the escape paths for participants $i_1,\dots,i_j$.
Therefore, a maximum flow corresponds to the maximum number of participants that can reach the exit.
Runtime: We can construct the Network in time linear in the size of the grid, which has n^2 vertices and $m \le n^2$ edges. Then we compute the maxflow on G' in time $T_{\text{maxflow}(n^2,m)}$ and return the resulting value. This totals a runtime of $\mathcal{O}\big(T_{\text{maxflow}}(n^2,m)\big)$.

c. After you posted your very successful algorithm online, the shows' network is quite unhappy: people are no longer interested in the game as your algorithm spoils the answer too easily! To keep it interesting, they devise a new rule: As soon as a player leaves a room, all doors of that room are locked. **(_ /4 P.)**
Design an algorithm that determines the maximum number of participants that can reach the exit with the new ruleset. Again, your algorithm should run in time $\mathcal{O}\big(T_{\text{maxflow}}(n^2,m)\big)$ with the same input. Prove the correctness and analyze the runtime of your algorithm.

Solution

We can model new ruleset with the same network, but this time we need node capacities: Locking the doors after leaving a room is equivalent to the paths being vertex-disjoint. By introducing unit node-capacities into our flow-network, we ensure that all paths are vertex disjoint, as only a single unit of flow can be pushed through any vertex – this corresponds to the new rule, disallowing a room to be accessed after a participant left it.
To model this as a general flow network, we replace every vertex v by $(v_{\text{in}}, v_{\text{out}})$ connected by a unit capacity edge, connecting ingoing edges to v_{in}, outgoing edges to v_{out}. To see that this is equivalent to node capacities, it is sufficient to see that any flow going into v_{in} can only be pushed further through v_{out}, thus being subject to the capacity of the edge between. The correctness argument stays the same, except that the paths in the maze are now vertex disjoint and flows in the network are vertex disjoint as well. Thus, the equivalence between then number vertex disjoint paths and the maximum flow value of our node-capacitated network is maintained.
The new network still has $\mathcal{O}(n^2)$ vertices and $\mathcal{O}(m) = \mathcal{O}(n^2)$ edges, thus the runtime consideration stays the same as well.

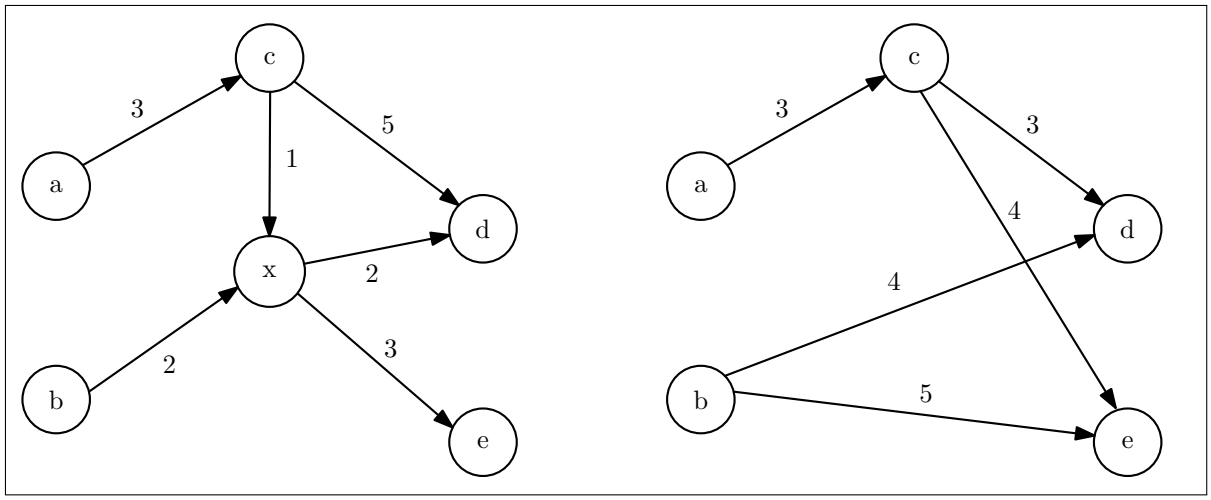
Sample Solution

Problem 3. Shortest Paths: Pairwise Shortest Paths **(_ /9 P.)**

In the following we will work with an edge-weighted, directed graph $G = (V, E, c)$ with n vertices, m edges and positive edge weights $c : E \to \mathbb{N}$. We denote by $d_G(u, v)$ the length of the shortest path in G between vertices u and v.

a. Consider the edge-weighted graph given below. Remove vertex x and draw the resulting graph such that the length of all shortest paths is maintained for all $u, v \in V \setminus \{x\}$. **(_ /1 P.)**

Solution



b. Given $G = (V, E, c)$, design an algorithm that constructs a graph $G_x = (V \setminus \{x\}, E', c')$ such that $d_G(u,v) = d_{G_x}(u,v)$ for all $u, v \in V \setminus \{x\}$. Your algorithm should run in time $\mathcal{O}(n^2)$. Prove the correctness and analyze the runtime of your algorithm. **(_ /3 P.)**

Solution

We maintain length of the shortest paths, by effectively shortcutting vertex x. More precisely, we replace every incoming edge to x by several shortcut edges that connect to all vertices that x could reach. The weight of a shortcut edge is now given by the sum of the original two edges. We avoid multi-edges by only considering the smaller weight edge between two vertices.
We define

$$E' = \{(u,v) \mid (u,x) \in E, (x,v) \in E\} \cup \{(u,v) \in E \mid u \neq x \wedge v \neq x\}$$

with weights

$$c'(u,v) = \min\{c(u,v), c(u,x) + c(x,v)\}$$

for $u, v \in V \setminus \{x\}$ (for brevity we assume that the weight of non-existent edges is ∞).
Then return $G_x = (V \setminus \{x\}, E', c')$.
Correctness: Consider the shortest path $P = s, \ldots, t$ between two vertices $s, t \in V$ in G of length $d_G(s,t)$. If this path does not go through x, then the length stays the same in G_x, as $w(u,v) \leq w(u,x) + w(x,v)$ for all u, v contained in P.
Assume now that $P = s, \ldots, u, x, v, \ldots, t$ goes through x and consider its length

$$d_G(s,t) = d_G(s,u) + c(u,x) + c(x,v) + d_G(v,t).$$

By our construction, there thus exists a path $P' = s, \ldots, u, v, \ldots, t$ in G_x of length

$$\begin{aligned} d_{G_x}(s,t) &= d_G(s,u) + c'(u,v) + d_G(v,t) && (1)\\ &= d_G(s,u) + c(u,x) + c(x,v) + d_G(v,t) && (2)\\ &= d_G(s,t). && (3) \end{aligned}$$

Runtime: The new edge set is of size $\deg^+(x) \cdot \deg^-(x)$, and thus of size at most $\mathcal{O}(n^2)$. Computing the weights can be done in time linear in the number of edges. We thus spend time at most $\mathcal{O}(n^2)$ constructing G_x.

c. Given G and a vertex x, let G_x be as described in the previous exercise. Design an algorithm that, given $G=(V,E,c)$, $x \in V$ and the pairwise shortest distances d_{G_x} of G_x, computes all shortest distances from x to $u \in V \setminus \{x\}$. Your algorithm should run in time $\mathcal{O}(n^2)$. Prove the correctness and analyze the runtime of your algorithm. **(_ /3 P.)**

Solution

By property of the previous algorithm, it is ensured that $d_G(u,v) = d_{G_x}(u,v)$ for all $u,v \in V \setminus \{x\}$. Thus, it is only left to compute $d_G(x,v)$ and $d_G(v,x)$ for all $v \in V$.
It is sufficient to compute

$$d_G(x,v) = \min_{(x,u)\in E} c(x,u) + d_{G_x}(u,v)$$

as we are promised that $d_{G_x}(u,v) = d_G(u,v)$, and thus the re-introduction of x cannot improve any distances between pairs of vertices compared to d_{G_x}, which suffices to argue **correctness**.

Runtime: We need to compute for every $v \in V \setminus \{x\}$ the shortest distances over vertices from $V \setminus \{x\}$, taking time at most $\mathcal{O}(n^2)$ as $d_{G_x}(u,v)$ is already provided for every pair of vertices $u,v \in V \setminus \{x\}$.

d. Denote the algorithms from part **b.** and **c.** as B and C. Can you get $\mathcal{O}(n^{2-\varepsilon})$-time algorithms with $\varepsilon > 0$ for B and C? Otherwise, show that this is unlikely under one of the fine-grained hypotheses introduced in the lecture. **(_ /2 P.)**

Hint: You may use without proof that the following algorithm correctly computes all pairwise shortest distances d_G in a given graph G:

If G contains a single vertex v then return $d_G(v,v) = 0$.
Otherwise:

1. *Pick some arbitrary vertex $x \in V$*
2. *Compute G_x with algorithm B, removing vertex x.*
3. *Recurse on the graph G_x to compute all pairwise distances d_{G_x}.*
4. *Use algorithm C with d_{G_x} to compute the all pairwise distances d_G.*

Solution

We prove a conditional lower bound under the APSP hypothesis. The base case can be computed in constant time, the given algorithm recurses n times in total and every recursion step takes $\mathcal{O}(T_B + T_C)$ resulting in a runtime of $\mathcal{O}(n(T_B + T_C))$. Now, if B and C had $\mathcal{O}(n^{2-\varepsilon})$-time algorithms for some $\varepsilon > 0$, our APSP algorithm would run in time $\mathcal{O}(n^{3-\varepsilon})$, refuting the APSP Hypothesis.

Sample Solution

Problem 4. Randomized: Sibling Sabotage **(_ /9 P.)**

You are a world-class music producer. You have just finished processing a raw audio sample vector $x \in \mathbb{F}_2^n$ through your signature digital effects matrix $M \in \mathbb{F}_2^{n \times n}$ to create the final track $y = Mx$ i.e., y is the matrix-vector product of M and x.
However, your little brother is furious that you didn't take him to the local carnival. In an act of vengeance, he accesses your workstation. He either does not find your masterpiece y, or he trashes it significantly. You are **promised** that the resulting vector y' follows one of two cases:

- **Case 1 (The Masterpiece):** $y' = Mx$.
- **Case 2 (The Sabotage):** At least $n/2$ entries are changed (in other words: the Hamming-Distance is $d_H(Mx, y') \geq n/2$).

The task is: Given M, x and y' but in particular not y, you want to decide which case occurred without recomputing $y = Mx$.

a. Assume you want to be 100% certain whether the track is Case 1 or Case 2. Prove that any deterministic algorithm must, in the worst case, examine at least $\Omega(n)$ entries of the vector y'. **(_ /2 P.)**
Hint: Consider an adversary who only modifies bits that your algorithm has not yet queried.

Solution

As in the worst case only $n/2$ many entries were changed, an adversary could place the $n/2$ remaining unchanged bits in such an order, that the deterministic algorithm examines exactly those entries first. Thus, only after $n/2 = \Omega(n)$ checks can the deterministic algorithm determine that the masterpiece y' is sabotaged, even if we were to know the (original) masterpiece.

Sample Solution

b. Design a randomized algorithm that: **(_ /4 P.)**

1. Always outputs the correct classification.
2. Has an **expected** runtime of $\mathcal{O}(n)$ when the track is sabotaged.

Prove the correctness and analyze the (expected) runtime of your algorithm.

Solution

We iterate up to $n/2$ times:

1. Pick an index $i \in [n]$ uniformly at random *without replacement*.
2. Compute the actual value of y_i by multiplying the i-th row of M with x. This takes $\mathcal{O}(n)$ time.
3. Compare y_i to the given y'_i. If $y_i \neq y'_i$, immediately output that the track is sabotaged and terminate.

If no discrepancy is found after $n/2$ iterations, output that the track has not been sabotaged.
Correctness:
If the masterpiece is completely intact, $y_i = y'_i$ for all i, so the algorithm correctly accepts after $n/2$ iterations. If the track is sabotaged, strictly more than $n/2$ entries must have been altered. Since we sample $n/2$ distinct indices without replacement, but there are at least $n/2$ many modified bits in the sabotaged case, we are guaranteed to check a modified bit. Thus, the algorithm always correctly classifies the input.
Expected Runtime (Sabotaged Case):
When the track is sabotaged, the probability p of discovering a tampered bit in any given iteration is at least $\frac{1}{2}$ (the probability only increases as we sample without replacement). To give an upper bound on the runtime, it is sufficient to model these checks as independent trials with success probability $p \geq \frac{1}{2}$. This follows exactly the geometric distribution.
We calculate the expected number of iterations $\mathbb{E}[I]$ as:

$$\mathbb{E}[I] \leq \sum_{k=1}^{\infty} P[\text{tampered bit found in iteration } k] \tag{4}$$

$$\leq \sum_{k=1}^{\infty} k \cdot (1-p)^{k-1} p = \frac{1}{p} \leq 2 \tag{5}$$

Since computing the scalar dot product for each iteration takes $\mathcal{O}(n)$ time, the total expected runtime is bounded by $\mathbb{E}[I] \cdot \mathcal{O}(n) = 2 \cdot \mathcal{O}(n) = \mathcal{O}(n)$.

Sample Solution

c. The record label is demanding the file immediately. You switch to a **Monte Carlo** approach. You will sample k indices $\{i_1, i_2, \ldots, i_k\}$ independently and uniformly at random from $\{1, \ldots, n\}$. You consider the file y' the (untampered) Masterpiece if it holds that $(Mx)_{i_j} = y'_{i_j}$ for all $1 \leq j \leq k$. Otherwise, you consider the track sabotaged. **(_ /3 P.)**

1. Determine the minimum number of samples k required such that the probability of a False Positive (claiming the track is intact when it is actually sabotaged) is less than $1/1000$.
2. Show that there is an $\mathcal{O}(n \log n)$-time algorithm determining whether the track is sabotaged with error probability at most $\frac{1}{n}$.

Solution

1. As the sample are chosen independently and uniformly at random and in the worst case only $n/2$ bits are tampered, we have a probability of $\frac{1}{2}$ that a single sample will detect the sabotage. For a false positive to occur, we thus need to pick a non-tampered bit k times, which happens with probability

$$\left(\frac{1}{2}\right)^k \leq \frac{1}{1000}.$$

   $k = 10$ is therefore the minimal k such that the probability of a false positive is $1/1024 < 1/1000$.

2. The given Monte Carlo algorithm can easily be modified into a $\mathcal{O}(n \log n)$ algorithm with $1/n$ error probability by choosing $k = \lceil \log n \rceil$. The probability of a false positive then is

$$\left(\frac{1}{2}\right)^{\lceil \log n \rceil} \leq 1/n.$$

   As computing $(Mx)_{i_j} = y'_{i_j}$ for $j \in [k]$ takes time $\mathcal{O}(n)$ we obtain a $\mathcal{O}(n \log n)$ total running time.

Sample Solution

Problem 5. Geometric Algorithms: Asteroids **(_ /9 P.)**

In the basement of your grandparents' house, you found an old arcade machine, loaded up with a bootleg version of the famous "Asteroids" game. You control a spaceship equipped with powerful blasters in a 2-dimensional plane and your goal is to shoot down asteroids. Because this is a bootleg version, it works slightly differently:

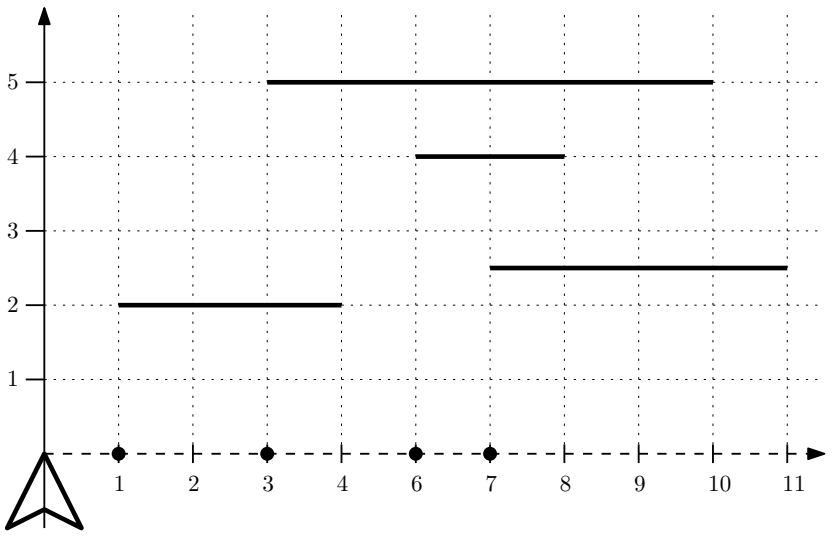
- The ship may only move along the x-axis (its y-coordinate is always 0).
- Every asteroid is a static line segment parallel to the x-axis, i.e. $A_i = \overline{(\ell_i, y_i)(r_i, y_i)}$ with $y_i \geq 1$. The start and endpoint are included in the line segment.
- A shot fired from $(x, 0)$ travels instantly along the vertical line $\{x\} \times \mathbb{R}$ and only hits and destroys the first asteroid it intersects.

The spaceship starts at the origin $(0, 0)$.

a. Given the configuration below, give a sequence of x-coordinates from which to fire such that every asteroid is hit exactly once. **(_ /1 P.)**

Solution

One possible solution could look like this:



b. Let $x_1 < x_2 < \cdots < x_n$ be a **monotone** sequence of firing positions and let $A_1, \ldots, A_m$ be the asteroids, each given as a segment $\overline{(\ell_i, y_i)(r_i, y_i)}$ with $y_i \geq 1$. **(_ /4 P.)**
Design an $\mathcal{O}(n \log n)$-time algorithm that decides whether all asteroids are destroyed. Prove the correctness and analyze the runtime.

Solution

First, let us note that if there are $m > n$ many asteroids, then, we can not hope to destroy them all with only n shots. W.l.o.g. we can thus assume that $m \leq n$.
Idea. We devise a sweep-line algorithm going from the left to the right. Begin by sorting the asteroids by their start-point in x-direction. We use an addressable PQ as data structure, inserting the asteroids based on their y-coordinate.
Events: Events are the start and end points of the asteroids as well as our firing positions. We process them in the following order and manner:

- When scanning a start-point for asteroid A_i, we add (i, y_i) to the priority queue, ordered by y_i.
- When scanning a firing position, consider the minimum asteroid A_i of the PQ. We remove it from the PQ.
- When we scan an end point for asteroid A_i, we check if an element with key i exists in the PQ. If so, the given sequence can not destroy all asteroids and we abort.

If we finish the sweep without preemptively determining that there is an asteroid that we cannot destroy, the given firing sequence destroys all asteroids.
Correctness: We keep track of all active asteroids, reducing the health the asteroid with the lowest y coordinate, as the shot travels from $y = 0$ and only hits the first asteroid intersected. As we reduce the health value per hit correctly and only remove the asteroid from the PQ if its health hits 0 we correctly determine when an asteroid is destroyed. If we scan an end point for an asteroid that has not yet been removed, then by monotonicity of firing positions, that no more shots will hit it thus can abort preemptively.
Thus, if the sweepline reaches the end without aborting preemptively, the sequence reduces every single asteroid's health to 0.
Runtime: Sorting the asteroids can be done in $\mathcal{O}(n \log n)$ as $m \leq n$. There are $\mathcal{O}(n)$ many events and handling an event can be done in time $\mathcal{O}(\log n)$ at most. This results in a $\mathcal{O}(n \log n)$-time algorithm.

c. After collecting some upgrades, your ship can place a powerful space bomb that explodes in a $W \times W$ square, *annihilating* and completely destroying any asteroid that is fully contained in that $W \times W$ square. A space bomb can be described by its bottom left corner (s_x, s_y), where the area covered is given by the square $[s_x, s_x + W] + [s_y, s_y + W]$. **(_ /4 P.)**

Your goal is to find the perfect position to drop the bomb, as you want to make the most use of it. Assume for the following that all asteroids are of fixed length $\alpha < W$.

Given n positions $S_1, \ldots, S_n$ for $W \times W$ space bombs and n asteroids $A_1, \ldots A_n$, each given as a segment $\overline{(\ell_i, y_i)(r_i, y_i)}$ with $y_i \geq 1$ and $r_i - \ell_i = \alpha$, Design an $\mathcal{O}(n \log n)$-time algorithm that determines the maximum number of asteroids you can *annihilate* by placing the space bomb at some position S_i with $i \in [n]$. Prove the correctness and analyze the runtime.

Solution

Idea: Because every asteroid has length α, it is fully inside a bomb's $W \times W$ square iff its midpoint lies in a reduced square of side $W - \alpha$.

Preprocessing:

1. For each asteroid $A_i = \overline{(\ell_i, y_i)(r_i, y_i)}$ compute its center $P(A_i) = \big(c_i, y_i\big)$ with $c_i = (\ell_i + r_i)/2$.
2. Insert all $P(A_i)$ into a 2-D orthogonal range search data structure. Construction costs $\mathcal{O}(n \log n)$ time and space.

For a bomb $S_j = (s_x, s_y)$ the explosion covers $B_j = [s_x, s_x + W] \times [s_y, s_y + W]$. An asteroid is hit if and only if its center lies in

$$Q_j = [s_x + \tfrac{\alpha}{2}, s_x + W - \tfrac{\alpha}{2}] \times [s_y + \tfrac{\alpha}{2}, s_y + W - \tfrac{\alpha}{2}].$$

Perform a range-count query on Q_j; the result equals the number of asteroids hit by S_j.

Algorithm:

1. Build the range search data stucture on $\{P(A_i)\}$ $\mathcal{O}(n \log n)$
2. For each bomb S_j $(j = 1 \ldots n)$:
   - Compute Q_j. $\mathcal{O}(1)$
   - Query the data stucture for $|Q_j|$. $\mathcal{O}(\log n)$
   - Keep the maximum over all queries. $\mathcal{O}(1)$
3. Output the maximum.

Correctness: It is sufficient to show that an asteroid is hit if its center lies in Q_j. Observe that A_i is hit $\Rightarrow s_x \leq \ell_i$, $r_i \leq s_x + W$. Using $\ell_i = c_i - \alpha/2$, $r_i = c_i + \alpha/2$ gives $s_x + \alpha/2 \leq c_i \leq s_x + W - \alpha/2$; similarly for y_i. Hence $P(A_i) \in Q_j$. The converse follows by reversing the algebra.
Therefore, the range-count query thus returns exactly the number of hit asteroids.

Runtime: Transforming segments the line segments takes $\mathcal{O}(n)$ plus an additional $\mathcal{O}(n \log n)$ time to construct the orthogonal range search data structure. We perform in total n many queries running in time $\mathcal{O}(\log n)$ The overall running time thus is $\mathcal{O}(n \log n)$.

Sample Solution

Problem 6. Stringology: Evenly Spaced Patterns **(_ /11 P.)**

Recall the Pattern Matching problem introduced in the lecture. We want to find more general patterns now: Given a binary string $x \in \{0,1\}^n$ of length n, determine whether there exists a pattern of three evenly spaced 1s, i.e. such that they are separated by the same number of characters.

Examples: **1**0**1**0**1**, **111**00 or **1**00**1**10**1**.
Non-Examples: 101001 or 11010001.

a. State whether the string 0001011011 contains three evenly spaced 1s. **(_ /1 P.)**

b. Prove the following claim: **(_ /2 P.)**

A string $x[1 \dots n]$ contains a pattern of three evenly spaced 1s if and only if there exist indices $i < j < k \in [n]$ with $i + k = 2j$ such that $x[i] = x[j] = x[k] = 1$.

Solution

As the ones are evenly spaced it has to hold that $x[a] = x[a+b] = x[a+2b] = 1$. Thus, there exist $i = a, j = a+b$ and $k = a+2b$ with $i+k = 2$ that satisfy the claim. Conversely, consider a triple (i, j, k) such that $x[i] = x[j] = x[k] = 1$ with $i + k = 2j$. By subtracting i and j we get that $k - j = j - i$, thus the ones are evenly spaced, proving the claim.

c. Design an algorithm that determines whether there occur three evenly spaced 1s in a binary string x of length n in time $\mathcal{O}(n \log n)$. Prove the correctness and analyze the runtime of your algorithm. **(_ /5 P.)**

Hint: use the claim from b)

Solution

Convert the string x into a polynomial p of degree n where

$$p(y) = \sum_{i \in [n] : x[i] = 1} y^i.$$

We use FFT to compute $q(y) = p(y)^2$ in time $\mathcal{O}(n \log n)$ as p is of degree n.
The resulting polynomial $q(y)$ has the following properties: Consider monomial y^{2l}. It can be produced in two ways: either p contained the monomial y^l or it contained two distinct monomials y^i and y^k with $i + k = 2l$. As we constructed our polynomial such that y^i has coefficient 1 only if $x[i] = 1$, it is sufficient to check whether the coefficient of the degree l monomial in $q(y)$ larger than one, as this implies that $x[l] = 1$ as well as that $i \neq k$ with $i + k = 2j$ exists such that $x[i] = x[k] = 1$. Using concludes that three evenly spaced ones exist.
Runtime: Polynomial can be constructed in $\mathcal{O}(n)$ while FFT takes time $\mathcal{O}(n \log n)$ for degree n polynomials. Finally, we can check all coefficients of q in time $\mathcal{O}(n)$ as well.

d. Let Σ be a finite alphabet and $W_1, W_2 \in \Sigma^k$. We are interested in detecting the pattern $P = W_1 ?^k W_2$ where '?' denotes a *wildcard* that matches any character of the alphabet Σ, i.e. W_1 and W_2 are separated by any k characters. **(_ /3 P.)**
Given a string $x \in \Sigma^n$, a number $k \in \mathbb{N}$ and $W_1, W_2 \in \Sigma^k$, design an algorithm running in time $\mathcal{O}(n+k)$ that determines whether the pattern $P = W_1 ?^k W_2$ occurs in x. Prove the correctness and analyze the runtime of your algorithm.

Example: $\underbrace{\texttt{110}}_{W_1}\underbrace{\texttt{010}}_{?^3}\underbrace{\texttt{101}}_{W_2}\texttt{01}$ contains $P = \texttt{110???101}$.

Solution

Use KMP to report all matches for W_1 and W_2 in time $\mathcal{O}(n+k)$ each. Check if there exist two matches for W_1 and W_2 that are exactly k apart.
Correctness: KMP finds all occurrences of W_1 and W_2 correctly. Thus checking the existence of two matches being apart k characters is sufficient for the existence of P.
Running Time: Two runs of KMP can be done in time $\mathcal{O}(n+k)$. Using i.e. arrays of length n to track the matches, we two matches that are k apart in time $\mathcal{O}(n)$. The total run time is thus $\mathcal{O}(n+k)$.