University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn

Exam Algorithm Theory

Wednesday, March 28, 2018, 9:00-11:00

Name: ....

Matriculation No.: ....

Signature: ....

Do not open or turn until told so by the supervisor!

- Please put your **student ID** on the table next to you so we can check it.
- Write your **name** and **matriculation number** on this page and sign the document!
- Your **signature** confirms that you have answered all exam questions without any help, and that you have notified exam supervision of any interference.
- You are allowed to use a summary of **five (single-sided) A4 pages**.
- Write legibly and only use a pen (ink or ball point). Do **not use red**! Do **not use a pencil**!
- You may write your answers in **English or German** language.
- **No electronic devices** are allowed.
- Only **one solution per task** is considered! Make sure to strike out alternative solutions, otherwise the one yielding the minimal number of points is considered.
- **Detailed steps** might help you to get more points in case your final result is incorrect.
- The keywords **Show...**, **Prove...**, **Explain...** or **Argue...** indicate that you need to prove or explain your answer carefully and in sufficient detail.
- The keywords **Give...**, **State...** or **Describe...** indicate that you need to provide an answer solving the task at hand but without proof or deep explanation (except when stated otherwise).
- You may use information given in a **Hint** without further explanation.
- **Read each task thoroughly** and make sure you understand what is expected from you.
- **Raise your hand** if you have a question regarding the formulation of a task.
- A total of **45 points** is sufficient to pass this exam.
- A total of **90 points** is sufficient for the best grade.
- There is a **separate solution page** for each exercise and two additional **blank pages at the end**.
- Write your name on **all sheets**!

Question	1	2	3	4	5	**Total**
Points						
Maximum	37	14	15	26	28	120

Task 1: Short Questions (37 Points)

(a) *(4 Points)* Explain (briefly) what dynamic programming is and what properties a problem needs to have in order to admit an efficient dynamic programming algorithm.

(b) *(6 Points)* Given a **sorted array** $A[1 \dots n]$ containing n **distinct** integers (no integer occurs twice), we want to determine whether A contains a **fixed point**. I.e., is there an index $i \in \{1, \dots, n\}$ with $A[i] = i$.

Describe an algorithm that runs in time $\mathcal{O}(\log n)$ and outputs the index i of a fixed point if it exists, and otherwise 0.

(c) *(5 Points)* Given a **weighted, undirected** graph G, show that an edge e with minimum weight is always part of **at least one** minimum spanning tree of G.

(d) *(8 Points)* Consider the **PRAM model** with n processors. Assume that the memory cells $c_1, \dots, c_n$ contain nodes of a **forest of rooted trees**.

Roots are represented by memory cells c_r that contain their own number, i.e. $c_r = r$. **Non-root nodes** are represented by cells c_i that contain the number of their parent cell, i.e. $c_i = j$.

We propose the following parallel algorithm that is repeated several times: Processor i reads cell $c_i = j$, i.e., it reads the value j from c_i. If $i \neq j$ it writes $c_i \leftarrow c_j$ (write the content of cell c_j into cell c_i).

Argue why one repetition of the above parallel algorithm can be executed in constant time on a CREW PRAM. Also argue why this is not possible in constant time on an EREW PRAM.

Let the maximum height of any tree be h. Show that after $\mathcal{O}(\log h)$ repetitions of the above algorithm **by every processor**, every cell c_i contains the root of the tree it belongs to.

(e) *(6 Points)* Let $U, V \subseteq \mathbb{Z}^d$ be two sets each containing $\boldsymbol{k}$ d-dimensional **integer** vectors. The goal is to pair these vectors into k disjoint pairs $(\vec{u}_1, \vec{v}_1), \dots, (\vec{u}_k, \vec{v}_k) \in U \times V$ (each vector of U is paired with exactly one vector of V) such that the pairing has **minimum total cost**.

The **cost of a pair** $(\vec{u}, \vec{v}) \in U \times V$ is $\sum_{i=1}^{d} \max\{u_i, v_i\}$, where $\vec{u} = (u_1, u_2, \dots, u_d)$ and $\vec{v} = (v_1, v_2, \dots, v_d)$. The **total cost is the sum of the costs of all $\boldsymbol{k}$ pairs**. Show how to solve the problem optimally in polynomial time.

Hint: *You can use algorithms from the lecture.*

(f) *(8 Points)* Let $G = (V, E)$ be an **undirected** graph with n nodes, where **every node $\boldsymbol{v} \in \boldsymbol{V}$ has degree at least $\boldsymbol{c \ln n}$ for some constant $\boldsymbol{c > 0}$**. We propose to select a subset of nodes $V' \subseteq V$ by flipping a fair coin for each node. That means with probability $\frac{1}{2}$ we add a given node $v \in V$ to V'.

Let G' be the sub-graph of G induced by the nodes in V'. Show that for an appropriate choice of the constant $c > 0$, the degree of every node v in G' is at most $(\frac{1}{2} + \frac{1}{5})d(v)$ with probability at least $1 - \frac{1}{n}$.

Sample Solution

(a) *Short version:* "Dynamic Programming $\approx$ Recursion + Memoization" (sufficient).

In a sentence: A problem is recursively broken down into sub-problems, whose solution is stored for future reference in case that sub-problem needs to be solved several times during the recursion.

Grading: 1 point for mentioning recursion. 3 points for either mentioning the keyword memoization, or explaining that a problem is broken down into sub-problems whose solutions are stored. Alternatively one can argue that a table is used to store solutions of sub-problems (calculated with previous results) bottom up in that table.

(b) We find a fixed point using a simple divide and conquer approach. Initially we set $l \leftarrow 1$ and $r \leftarrow n$. If $l < r$ we pick a pivot $p \leftarrow \lfloor \frac{r+l}{2} \rfloor$ and check whether $A[p] = p$, in which case we return p.

Else, if $A[p] > p$ then a fixed point can only be in the left half of A (since A is sorted and contains distinct integers), hence we set $r \leftarrow p - 1$ and do a recursive step on $A[l \dots r]$.

Analogously, if $A[p] < p$ a fixed point must be in the right half of A if it exists and we set $l \leftarrow p + 1$ and do a recursive step. In the base case we have $r < l$ and we return 0.

Remark: This approach achieves the required run-time: $T(n) = T(\frac{n}{2}) + \mathcal{O}(1) = \mathcal{O}(\log n)$. It is correct because the values in the array increase (when going from left to right) or decrease (going from right to left) at least as fast as the indices. We maintain the invariant that a fixed point can not be in the half we do not recurse on.

Grading: Emphasis is ***not*** *on the pointers l, r and how to compute those! Students need to mention the selection of a pivot element in the middle of the array A (1 point) and use it to decide on which half of the array A (e.g. left or right) the recursion should continue (4 points). Finally the two base cases needs to be mentioned: Fixed point found and partial array empty (1 point).*

Simply mentioning that the problem can be solved using binary search is not completely correct. E.g. we do not have an element to search for, instead we have to compare the pivot-pointer p with $A[p]$, i.e. the array at the position of the pivot-pointer. However, since the idea is very similar mentioning binary search and nothing else yields 2 points.

(c) Consider a MST T of G which does not contain e. Then $T \cup \{e\}$ contains a cycle C. Pick any edge $e' \in C$ that is not e. Then $\big(T \setminus \{e'\}\big) \cup \{e\}$ is a MST with at most the same weight T and must therefore be a MST.

(d) Every processor i only writes on its own register c_i thus there are never any concurrent writes to the same register. Hence a concurrent read exclusive write (CREW) PRAM can execute one repetition of the given algorithm in parallel in $\mathcal{O}(1)$ time with all n processors *(1 Points)*.

This is not possible with an exclusive read exclusive write (EREW) PRAM. Take for example a tree with a single root-cell $c_r = r$, where all other nodes are direct children of c_r, i.e. $c_i = r$ for all $i \neq r$. Here the read operation would have to take place in a sequential fashion taking $\mathcal{O}(n)$ time steps *(2 Points)*.

The claimed number of repetitions is true for $h = 1$ since in this case all nodes are roots or children of roots and we do not need to do anything. Consider a longest path $(i_0, \ldots, i_h)$ from a leaf c_{i_0} to a root c_{i_h} where $h > 1$ is the height of the tree. That means cell c_{i_k} contains i_{k+1} for $k < h$ and the root cell c_{i_h} contains i_h.

After one step of the given parallel algorithm we have $c_{i_k} = i_{k+2}$ for $k \leq h-2$ and cells $c_{i_h}, c_{i_{h-1}}$ both contain i_h. Hence in the new tree we now have two paths $(i_0, i_2, \ldots, i_h)$, $(i_1, i_3, \ldots, i_h)$ each consisting of at most $\lceil \frac{h+1}{2} \rceil$ cells and thus both have length at most $\frac{h}{2}$.

The algorithm is finished if the tree reaches height 1. This is the case after x rounds, if $\frac{h}{2^x} \leq 1$, i.e., $x \geq \log_2 h \in \mathcal{O}(\log h)$. Since a nodes grandparent has the same root as its parent, a node retains its original root in every repetition. *(5 Points).*

Grading: An informal explanation that paths from leafs to roots halve in length together with the explanation that the algorithm may stop if the height of the forest is 1*, is sufficient.*

(e) Construct a complete, bipartite graph where each node on the left hand side corresponds to a vector in U and each node on the right hand side represents a vector in V. Then each edge between vectors $\vec{u}_i \in U$ and $\vec{v}_j \in V$ has weight equal to the cost of pair $(\vec{u}_i, \vec{v}_i)$. Then calculate the *minimum weight perfect matching* for the constructed bipartite graph using the polynomial time algorithm from the lecture (we constructed a linear program, but the student does not need to mention this).

(f) Let v be a node of G'. Let X be the variable describing the random number of neighbors of v in G'. Let $\mu := \mathbb{E}(X) = \frac{d(v)}{2}$. We use the Chernoff bound (upper tail) to bound the probability that the degree is bigger than $(\frac{1}{2}+\frac{1}{5})d(v)$

$$\mathbb{P}\Big(X > (\frac{1}{2}+\frac{1}{5})d(v)\Big) = \mathbb{P}\Big(X > (1+\frac{2}{5})\mu\Big) < \exp\Big(-\frac{2^2 \cdot d(v)}{2 \cdot 3 \cdot 5^2}\Big) \leq \exp\Big(-\frac{2c \log n}{75}\Big) = n^{-\frac{2c}{75}}$$

(4 Points)

Let E_v be the event that node $v \in V$ has a degree bigger than $(\frac{1}{2}+\frac{1}{5})d(v)$. We use the union bound to upper bound the probability E_v happens for at least one node.

$$\mathbb{P}\Big(\bigcup_{v \in V} E_v\Big) \leq \sum_{v \in V} \mathbb{P}(E_v) = n \cdot \mathbb{P}\Big(X > (\frac{1}{2}+\frac{1}{5})d(v)\Big) = n^{1-\frac{2c}{75}}$$

For c big enough ($c \geq 75$) we have $1 - \mathbb{P}\Big(\bigcup_{v \in V} E_v\Big) \geq 1 - \frac{1}{n}$. *(4 Points)*

Task 2: Priority Queues and Prim's Algorithm (14 Points)

(a) *(9 Points)* Assume we want to store $(key, data)$-pairs in a priority queue where the priorities (keys) are only from the set $\{1, \dots, c\}$ and $c \in \mathbb{N}$ is constant.

Describe a **priority queue** that provides the operations `Insert`$(key, data)$, `Get-Min`, `Delete-Min`, and `Decrease-Key`$(pointer, newkey)$ all in **constant time** for the given scenario and **describe how these operations work** on your data structure.

(b) *(5 Points)* State how fast Prim's algorithm to compute a minimum spanning tree is, under the assumption that edge weights are in the set $\{1, \dots, c\}$ and $c \in \mathbb{N}$ is constant, using your implementation of a priority queue. **Explain your answer**.

***Remark**: Assume you have a priority queue as in (a), even if you did not succeed in (a).*

Sample Solution

(a) We use an array $A[1, \dots, c]$ of size c where each array entry contains a reference to a doubly linked list of $(key, data)$-pairs.

`Insert`$(key, data)$: When we insert a pair $(key, data)$ we simply append it to the list in $A[key]$ in $\mathcal{O}(1)$.

`Get-Min`: We iterate the Array starting from the beginning (in $\mathcal{O}(c) = \mathcal{O}(1)$), until we find a non-empty list at index i. We return the first pair $(i, data)$ from that list.

`Delete-Min`: We iterate the Array starting from the beginning (in $\mathcal{O}(c) = \mathcal{O}(1)$), until we find a non-empty list at index i. We remove the first pair $(i, data)$ from that list and return it.

`Decrease-Key`$(pointer, newkey)$: Since we have a pointer to the $(key, data)$-pair in question, we can remove and change its key in $\mathcal{O}(1)$. Afterwards we reinsert it into the correct list also in $\mathcal{O}(1)$

Grading: 9 base points are granted for the idea to use an array of linked lists, but up to 2 points are subtracted for each operation that is described wrong, incomplete, or not at all. Using a heap based implementation like e.g. binary heaps or Fibonacci heaps does not solve this task as these have logarithmic run-time for at least one of the operations in question.

Partial points can be granted if there is an explanation that some of the given operations work in $\mathcal{O}(1)$ using a heap based priority queue (up to 1 point per operation and explanation that it is in $\mathcal{O}(1)$).

(b) Prim's Algorithm now runs in $\mathcal{O}(|E| + |V|)$ using our implementation of the priority queue. *(2 Points)*

The reason is that Prim's algorithm uses $\mathcal{O}(|E|)$ `Decrease-Key` operations and $\mathcal{O}(|V|)$ `Delete-Min`, `Get-Min` and `Insert` operations. *(3 Points)*

Grading: In a MST Problem we typically have a connected graph thus $\mathcal{O}(|V|) \subseteq \mathcal{O}(|E|)$ and therefore the answer $\mathcal{O}(|E|)$ would also be correct for the run-time of Prim's algorithm.

Task 3: Multistack (15 Points)

A multistack is a peculiar data structure that consists of an infinite series of stacks $S_0, S_1, \ldots$ where **stack S_i can hold up to 3^i elements**. We assume that the multistack only supports an operation `push`(x, i) that pushes an element x on stack S_i.

Whenever an attempt is made to push an element x onto a full stack S_i, **all the elements** in S_i are first removed from S_i and then pushed onto stack S_{i+1} to make room in S_i. If S_{i+1} gets full while doing so, we first recursively move all its members to S_{i+2}, and so on.

Subsequently, the element x is pushed onto S_i. Moving a single element from one stack to the next one takes $\mathcal{O}(1)$ time.

(a) *(7 Points)* State the **worst case running time** to push one more element onto a multistack containing n elements. **Prove** your answer by means of a worst case example.

(b) *(8 Points)* Prove that the **amortized running time** of a push operation is $\mathcal{O}(\log n)$, if we push n elements onto the multistack in total.

Sample Solution

(a) Assume in the worst case that a maximum number of stacks $S_0, S_1, \ldots$ are full. By pushing one element to the first stack, the contents of each stack is recursively moved to the next higher stack. This means every element is moved exactly once, which leads to $\Omega(n)$ cost to push a single element into the multistack.

Grading: 2 points for stating that $\Omega(n)$ is the worst case running time (stating $\mathcal{O}(n)$ also counts). 3 points for giving a proper example where this time is achieved (maximum number of stacks $S_0, S_1, \ldots$ are full). 2 points for arguing why $\Omega(n)$ is the case in the example (cascading copies of stacks, each element moved once).

(b) In the worst case (with respect to maximum number of reinsert operations into the next stack) we insert always to the first stack. Assume that stacks $S_0, S_1, \ldots, S_k$ are all stacks with at least one element after the n inserts. Then each element is moved at most $k+1$ times. Since $k \in \mathcal{O}(\log n)$ we have our result. We see that $k \in \mathcal{O}(\log n)$ as follows

$$n \geq \sum_{i=0}^{k-1} 3^i = \frac{3^k - 1}{2} \quad \Longrightarrow \quad k \leq \log_3(2n+1) \in \mathcal{O}(\log n)$$

Grading: 3 points for describing the worst case scenario where we always push to the first stack. 5 points for the analysis that each element is moved at most $\mathcal{O}(k)$ times and $k \in \mathcal{O}(\log n)$ (stating $k \in \mathcal{O}(\log n)$ is sufficient).

Task 4: Two Sinks Maximum Flow Problem (26 Points)

We consider the following variant of the maximum flow problem. We are given a directed graph $G = (V, E)$ with **integer edge capacities** $c(e) > 0$ for all edges $e \in E$. There are three distinct nodes: A source node $s \in V$ and **two sink nodes** $t_1, t_2 \in V$.

The goal is to find a flow $f(e)$, $0 \leq f(e) \leq c(e)$ for every edge $e \in E$ such that for all nodes $v \in V \setminus \{s, t_1, t_2\}$, flow conservation holds, i.e., $f^{in}(v) = f^{out}(v)$. For the source and the sinks we require $f^{in}(s) = f^{out}(t_1) = f^{out}(t_2) = 0$, i.e., the source has only outgoing flow and the sinks have only incoming flow.

Further, the flow has to satisfy $f^{out}(s) = F$, $f^{in}(t_1) = f^{in}(t_2) = F/2$, for some value $F \geq 0$. That is, **the amount of incoming flow at t_1 and t_2 needs to be identical**. We call $F \geq 0$ the value of the flow f and we use F^* to denote the maximum possible flow value.

(a) *(5 Points)* Assume that the **intended** flow value $F \geq 0$ **is given as an input**. Further assume that F is an integer.

Describe an algorithm that determines whether the given flow network G admits a flow of value F.

Give the running time of your algorithm as a function of F and of the number of nodes n and the number of edges m of G.

(b) *(4 Points)* In the standard maximum flow problem, if all the capacities are integers, there is an **integral maximum flow**, i.e., a flow where all the **flow values $f(e)$ are integers**. In the above two sinks maximum flow problem, **this is not true**.

Give a flow network with one source s and two sinks t_1 and t_2 in which all capacities are integers, but where the maximum two sinks flow is **not integral**.

(c) *(9 Points)* Show that if all capacities are integers, the **maximum flow value F^*** of the above two sinks maximum flow problem is also an integer.

Hint: *Use the max flow min cut theorem.*

(d) *(8 Points)* Describe an algorithm to find a maximum flow of the two sinks flow problem. Give the running time of your algorithm as a function of F^* and of the number of nodes n and the number of edges m of G. Try to be **as efficient as possible**.

Hint: *You can use the fact that the optimal flow value F^* is an integer.*

Sample Solution

(a) Add a supersink t and two edges (t_1, t) and (t_2, t), each with capacity $F/2$ *(2-3 points)*. The two sinks max flow problem has a solution iff the s-t max flow problem has a maximum flow of value F.

The solution can be found by using the Ford Fulkerson algorithm *(1 point)*. All capacities of G are integers, however, the capacities of the two additional edges might only be half-integral. However, by scaling all capacities with a factor of 2, all capacities become integers *(0-1 point)*. Then we can solve the problem in time $O(mF)$, where m is the number of edges of G *(1 point)*.

(b) The network consists of four nodes s, v, t_1, and t_2. There are three edges (s, v), (v, t_1), and (v, t_2), all of capacity 1. Clearly the optimal flow value is 1 and thus the two edges (v, t_1) and (v, t_2) carry a flow value of $1/2$ each.

(c) Let F^* be the optimal flow value and consider the max flow reduction from (a), where the edges connecting t_1 and t_2 with the super sink t have capacity $F^*/2$.

For convenience, we define $V' := V \cup \{t\}$. This flow network has a maximum flow of value F^* and it therefore has a minimum s-t cut of value F^*. The cut $(V, \{t\})$ is such a minimum cut, however there needs to be another minimum cut $(S, V' \setminus S)$ as otherwise, F^* would not be optimal. (If the second smallest s-t cut has value $F^*+\delta$, we could increase F^* to $F^*+\delta$.)

There thus is an s-t cut of value F^* that contains at most one of the edges (t_1, t) and (t_2, t). Let C be the edges crossing that cut. If the cut does not contain one of the edges (t_1, t) and (t_2, t), it only consists of edges with integer capacities and we therefore get that F^* is an integer. Otherwise, it consists of an edge e (either (t_1, t) or (t_2, t)) of capacity $F^*/2$ and of integer capacity edges, hence

$$F^* = w(e) + \sum_{e' \in C, e' \neq e} w(e') \quad \Longrightarrow \quad F^*/2 = \sum_{e' \in C, e' \neq e} w(e')$$

and we thus even get that $F^*/2$ is an integer.

(d) We first find an upper bound $\hat{F} \geq F^*$ as follows. We solve the problem of (a) for $\hat{F} = 1, 2, 4, \ldots$ until the problem becomes infeasible *(3 points)*.

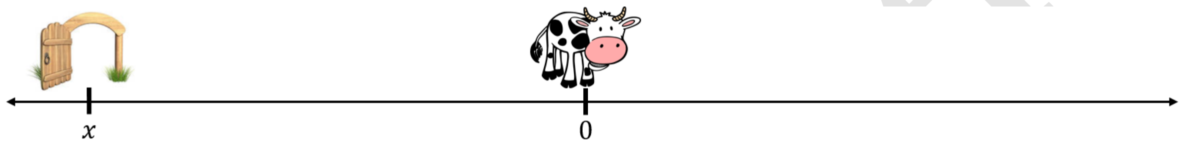
We thus have $\hat{F} > F^*$ and $\hat{F} \leq 2F^*$. We can now find the value of F^* (and an optimal flow) by using binary search between $\hat{F}/2$ and $\hat{F}$ *(3 points)*.

Overall, we need $O(\log F^*)$ iterations of the algorithm of (a), which has running time $O(mF^*)$. The overall running time is therefore $O(mF^* \log F^*)$ *(2 points)*.

Task 5: The Blind Cow Problem (28 Points)

Consider a blind cow facing a fence, infinite in both directions, attempting to find the only gate in the fence in order to get to the green pasture on the other side. Let us think of the fence as the real line where the cow is initially at position 0. The gate is positioned at position x on the line for some real value x **with** $|x| \geq 1$ (note that x might be negative or positive and thus the gate might be to the left or to the right).

The cow does initially not know x, it can only walk along the line and it always knows exactly how far it went. Since the cow is blind, it only realizes that it has reached the gate when it is at position x. The goal is to devise a travel strategy that minimizes the total distance that the cow needs to walk to find the gate.



(a) *(10 Points)* Give a **deterministic** algorithm for the blind cow to reach the gate. Your algorithm should be **9-competitive** (i.e., the total distance that the cow walks is at most $9 \cdot |x|$). **Prove** that your algorithm is 9-competitive.

Remark: *Partial points if can show that your algorithm is c-competitive for a constant $c > 9$.*

(b) *(5 Points)* Prove that no **deterministic** algorithm can be better than 3-competitive.

(c) *(6 Points)* Give an improved version of your deterministic algorithm from part (a) that uses **randomization**.

You can assume an oblivious adversary, i.e., the position x of the gate is determined before the algorithm starts to toss coins.

Argue why the **expected competitive ratio** of your randomized algorithm is strictly better than the **competitive ratio** of your deterministic algorithm.

(d) *(7 Points)* Use Yao's principle to show that no randomized algorithm can be better than 2-competitive.

Sample Solution

(a) The strategy is as follows. Consider the i^{th} hop as point $H_i = (-1)^i 2^i$ on the real line for $i = 0, 1, 2, 3, \ldots$. The cow moves to all the hops on a direct line one by one, i.e., it first goes to H_0, then H_1, H_2, and so on. *(5 Points)*

Let us assume for some integers $k, \epsilon \geq 0$, the gate is initially at position $|x| = 2^k + \epsilon$. Consider the following two cases:

(1) For even k and $x < 0$ or odd k and $x > 0$: The distance that the cows traverses is

$$1 + \sum_{i=1}^{k} 2 \cdot 2^i + 2^k + \epsilon < 5 \cdot (2^k + \epsilon)$$

(2) For even k and $x > 0$ or odd k and $x < 0$: The distance that the cows traverses is

$$1 + \sum_{i=1}^{k+1} 2 \cdot 2^i + 2^k + \epsilon < 9 \cdot (2^k + \epsilon)$$

(5 Points)

(b) Scenario description: Assume the adversary may only put the gate at -1 or 1. With this knowledge the best algorithm can only be more competitive. Considering the best deterministic algorithm $\mathcal{A}$, the adversarial strategy is as follows. If $\mathcal{A}$ visits $+1$ first, then the adversary puts the gate at -1 and otherwise at 1. *(3 Points)*

Analysis: Either way, with $\mathcal{A}$ the cow travels at least three times the distance than $|x|$. *(2 Points)*

(c) We use the same algorithm as in a) but the first movement of the cow is to the right with probability $1/2$ and else to the left. *(3 Points)*

Then with probability $1/2$ the traversed distance is the distance calculated in Case 1 of part (a) and with probability $1/2$ it is the calculated distance in Case 2 of part (a). Since the travel distance in Case 1 is strictly better than Case 2, the randomized algorithm is strictly better *in expectation* than the deterministic algorithm. *(3 Points)*

(d) Consider the following random input and the deterministic algorithm explained in part (a). With half probability the gate is at 1 and with half probability the gate is at -1. *(2 Points)*

Then, the expected traversed distance would be $\frac{1}{2} \cdot 3 + \frac{1}{2} \cdot 1 = 2$. *(2 Points)*

Using Yao's principle no randomized algorithm can do better than that, on a worst case input *in expectation*, so the competitive ratio is at least 2. *(3 Points)*