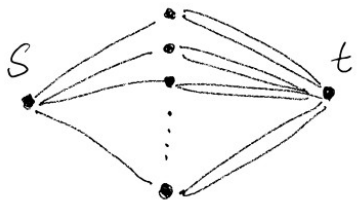
Problem 1.8

a

Consider the graph



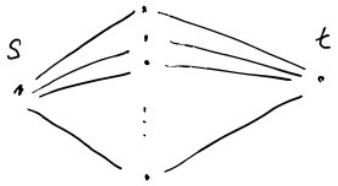
Removing any edge on the left side will increase the (s-t) min cut. Initially the left-side edges make up $\frac{1}{3}$ of the total amount of edges. At every stage if the algorithm has not contracted any wrong edge, the probability of contracting a wrong edge is than at least $\frac{1}{3}$. Thus we see that the probability of obtaining the min-cut is at most

$$\left(\frac{2}{3}\right)^{n-1}$$

Where n is the total number of edges. (This is exponentially decreasing in n).

b

Observe that any (s-t) min-cut define a unique split (2-partition) of the vertices. To see this first note that an (s-t) min-cut defines a split of the vertices defined by whether they are connected with s or not. Since it is a *cut* it must have removed any edge between the two groups of vertices. And since it is minimal, it cannot have removed any other edges. Thus it is unique. Now the number of splits of n vertices with s in the first group and t in the second is 2^{n-2}. So this is an upper bound on the number of min-cuts. Consider the graph



This achieves the upperbound of 2^{n-2} min-cuts, since any for any middle vertex you can choose either the left or the right edge to remove, resulting in a min-cut when done for all middle vertices.

Problem 2.1

We claim that for any deterministic evaluation algorithm there is a $T_{d,k}$ tree which forces the algorithm to evaluate all subnodes (and leafs), and furthermore we can choose the root node to evaluate to 0 and 1 as we please. Using DeMorgan it is easy to see that d-$\wedge$-$\vee$-trees are equivalent to d-$\downarrow$-trees so it is enough to consider these (we even allow the height to be odd). We proceed by induction on the height h of the tree. If $h = 0$ we are at leaf so we read $1 = d^0$ which we can choose to be 0 or 1. If $h = n$ its subtrees are of lower height. By induction we can choose the first $d - 1$ choices of subtrees of the algorithm to evaluate - using all nodes - to 0. This is because each zero forces the algorithm to evaluate more subtrees. The last chosen subtree of the algorithm we can choose to evaluate to 0 or 1 depending on whether we want the node to evaluate to 1 resp. 0. This completes the argument.

Exercise 1.2

Let H_n be the hand weight graph with $4n$ vertices, that is $2n$ on each side. At any point if the algorithm chooses two vertices on different sides, we increase the min-cut. After n succesful steps (that is steps not resulting in larger min-cut) each side still has at least n vertices left. The probability of choosing a pair from different sides during the first n steps is at least

$$\frac{\text{Number of pairs of vertices with each in different side}}{\text{Total number of pairs}} \geq \frac{n \cdot n}{4n \cdot (4n-1)/2} = \frac{n}{8n-4} \geq \frac{1}{8}$$

Exercise 1.2

Consider a graph with n vertices where the vertices can be partitioned into two subsets, S and T, of size $n/2$ (we assume for simplicity that n is even), such that the subgraphs consisting of these vertices and the edges between them are both complete.
Furthermore there are two vertices, one in each subgraph, call them s and t resp., that are connected by an edge, and no other vertex is connected to any other vertex of the other subgraph.
We call such a graph a *handweight graph*.

Clearly a handweight graph has a unique min-cut of size 1, performed by cutting the edge connecting s and t.
Furthermore, if a vertex from S is merged with a vertex from T the min-cut size is increased.

Exercise 1.3

Let V be the verifier algorithm outputting 0 if the solution is correct and 1 otherwise. Then the algorithm runs on an instance, I, in the following steps:

$v \leftarrow 0$
while $v = 0$
    $s \leftarrow A(I)$
    $v \leftarrow V(s)$
return s

This procedure clearly returns a correct solution if it terminates, though, a priori, it is not guaranteed that it terminates.

Note that at each iteration of the while loop A returns a correct solution with probability γ, hence the probability that the verifyer returns 1 is γ.
Since the algorithm exists the while loop the first time a correct solution is found, i.e. the first time the verifier returns 1, the number of iterations in the while loop is clearly geometrically distributed with success probability γ.
Hence we know that the expected number of iterations on input of size n is $1/\gamma(n)$, and each iteration clearly takes $T(n) + t(n)$.
Thus the expected time cost of the above procedure is $\frac{T(n)+t(n)}{\gamma(n)}$.

Problem 1.1

(a)

Let X_k denote the result of the kth coin toss with the biased coin and define the random variable

$$\tau = \min\{k \in \mathbb{N} \mid X_{2k-1} \neq X_{2k}\}$$

Now we define

$$Y = X_{2\tau-1}$$

We claim that the random variable Y represents a fair coin toss.

Indeed, by Baye's law and the independence of the biased tosses, we have for any $n \in \mathbb{N}$

$$\begin{aligned} P(X_{2n-1} = H \mid X_{2n-1} \neq X_{2n}) &= \frac{P(X_{2n-1} = H, X_{2n-1} \neq X_{2n})}{P(X_{2n-1} \neq X_{2n})} \\ &= \frac{P(X_{2n-1} = H, X_{2n} = T)}{P(X_{2n-1} \neq X_{2n})} \\ &= \frac{p(1-p)}{2p(1-p)} = 1/2 \end{aligned}$$

and likewise with tails.

To see that the expected number of tosses necessary to extract one fair toss with this procedure is $1/p(1-p)$ define

$$C_n = \begin{cases} 1, & \text{if } X_{2n-1} = X_n \\ 0, & \text{else} \end{cases}$$

Then $P(C_n = 1) = p^2 + (1-p)^2$ and $P(C_n = 0) = 2(p(1-p))$, so that if T denotes the number of tosses used we have for any $n \in \mathbb{N}$

$$P(T = 2n) = P(C_1 = 1, \cdots, C_{n-1} = 1, C_n = 0) = (p^2 + (1-p)^2)^{n-1} 2p(1-p)$$

So that the expected number of tosses is

$$\begin{aligned}
E(T) &= \sum_{n=1}^{\infty} 2n \left(p^2 + (1-p)^2\right)^{n-1} 2p(1-p) \\
&= 4p(1-p) \sum_{n=1}^{\infty} n \left(p^2 + (1-p)^2\right)^{n-1} \\
&= 4p(1-p) \frac{d}{dx} \left(\sum_{n=1}^{\infty} x^n \right) \Bigg|_{x=p^2+(1-p)^2} \\
&= 4p(1-p) \frac{1}{1-p^2-(1-p)^2} \\
&= 4p(1-p) \frac{1}{(2p(1-p))^2} \\
&= \frac{1}{p(1-p)}
\end{aligned}$$

as desired, where we have used that the power series converges for $|x| < 1$ and that $p^2 + (1-p)^2 < 1$ for $0 < p < 1$.

(b)

We first run the above described procedure on the list of biased flips until we reach the end of the list possibly disregarding the last flip if the number of flips is odd.
Every time we encounter a pair of different outcomes, we extract the unbiased coin flip as described and remove the pair from the list before we continue.
Now we are left with a list of pairs, where the outcomes of each pair are equal, and we create a new list by merging each pair into their common value.
Each entry in this new list is independent and distributed like the original flips, since the common value of a pair of tosses is determined by the value of the first. Hence we can run the procedure again on this new list, the length of which is at most half of that of the original list, extracting more unbiased coin tosses.
This procedure can be iterated until we are left with a list where all the values are equal, at which point we halt.

Problem 1.4

(a)

Since the permutations on n elements are in $1-1$ correspondence with the first $n!$ integers, we can pick any specific correspondence, $\varphi : \{1, \cdots, n!\} \to S_n$, use the random bits to generate a random integer, x, between 1 and $n!$ and compute $\varphi(x)$.
We would need $\lceil \lg(n!) \rceil$ bit to generate such an integer. Note that we might get an integer larger than $n!$, in which case we would have to try again until we get an integer in the desired range. The chance of getting an integer larger than $n!$ is at most $1/2$, so we expect getting an integer in the desired range after at most two tries.

Since there are $n!$ different permutations of n elements it takes $\lg(n!)$ bits to distinguish them all. So we would have to sample at least $\lg(n!) = \Omega(nlg(n))$ bits.

(b)

Consider two permutations $[i_1, \cdots, i_n]$ and $[j_1, \cdots, j_n]$. Then since the uniform variables are iid we have that the joint distribution of $(X_{i_1}, \cdots, X_{i_n})$ and $(E_{j_1}, \cdots, X_{j_n})$ are equal and hence

$$P\left(X_{i_1} < \cdots < X_{i_n}\right) = P\left(X_{j_1} < \cdots < X_{j_n}\right)$$

i.e., after sorting in ascending order, any two permutations of the variables are equally likely, hence they must all have probability $\frac{1}{n!}$ of occurring. So the indices of the sorted variables do indeed form a random permutation.

This permutation can be determined by sorting the random variables, e.g. by maintaining a list initialized to $[1, \cdots, n]$ and then performing the same operations on this list as on $[X_1, \cdots, X_n]$ when sorting it.

Treating the sampling of uniform variables as a black box, this scheme is as efficient as the sorting algorithm used to sort $[X_1, \cdots, X_n]$.

(c)

Problem 2.6

We will represent a deterministic sorting algorithm by a binary decision tree, each node of which represents a comparison made, its two branches represent the two possible outcomes of said comparison, and its two children represent the comparison it chooses to make based on the outcome.
I.e. the two children of a node represent the decision the algorithm makes based on the outcome of the comparison represented by the node. An execution of the algorithm is represented by following a path from the root down the tree, thus tracking the comparisons made, and the algorithm halts when a leaf is reached, i.e. there are no more decisions to be made and the list is sorted.
Hence the number of comparisons made by an execution of the algorithm is depth of the leaf representing its termination.
Since there are $n!$ different inputs, the tree must have $n!$ leaves.
When the input is chosen uniformly at random, the expected number of comparisons is then the average leaf-depth of its decision tree.

We proceed to show that any binary tree with k leaves has average leaf-depth at least $lg(k)$.
To this end consider any binary tree, T. For any node we will call the subgraph consisting of its two children and the two edges connecting it to its children its *branching*. Let h be the height of the tree and d the minimum depth of any leaf.
By *balancing* the tree we mean the process of choosing a leaf of depth $h-1$, removing its branching, and adding a branching at a leaf of depth d.
By iteratively balancing a T until $h-d \leq 1$ we obtain a completely balanced tree with k leaves.
We claim that if T is not completely balanced, any balancing reduces the average leaf-depth.
Indeed let S be the sum of the depth of all leaves of T, and S' the sum of the depth of all leaves of the tree obtained by performing a balancing on T. Then

$$S' = S - 2h + h - 1 - d + 2(d+1) = S - h + d + 1 \leq S$$

since the fact that T is not completely balanced yields that $h > d + 1$.
This proves that a completely balanced binary tree with k leaves minimizes the average leaf-depth of any binary tree with k leaves.

Since a completely balanced binary tree clearly has roughly $\lg(k)$ average leaf-depth (specifically between $\lfloor \lg(k) \rfloor$ and $\lceil \lg(k) \rceil$) this demonstrates the desired lower bound.

Returning to the decision tree of an arbitrary deterministic sorting algorithm, this proves that the average number of comparisons made by the algorithm on uniformly chosen input is at least $\lg(n!) = \Omega(n \lg n)$. Hence if p is the uniform distribution on $\mathcal{I} = \{[\pi(1), \cdots, \pi(n)] \mid \pi \in S_n\}$ and I_p denotes an input chosen according to this distribution then

$$\min_{A \in \mathcal{A}} EC(I_p, A) = \Omega(n \lg n)$$

where $\mathcal{A}$ is the (finite) set of deterministic sorting algorithms.
The assumption that this set is finite is readily seen to hold true since there are only finitely many binary trees with a fixed number of leaves.

Now Yao's principle yields that the expected running time of any las vegas algorithm is $\Omega(n \lg n)$.