Homework 1. Due October 11, 9:59PM.

CS181: Fall 2021

GUIDELINES:

- Upload your assignments to Gradescope by 9:59 PM.
- Follow the instructions mentioned on the course webpage for uploading to Gradescope very carefully (including starting each problem on a new page and matching the pages with the assignments); this makes it easy and smooth for everyone. As the guidelines are simple enough, bad uploads will not be graded.
- You may use results proved in class without proofs as long as you state them clearly.
- Most importantly, make sure you adhere to the policies for academic honesty set out on the course webpage. The policies will be enforced strictly. Homework is a stepping stone for exams; keep in mind that reasonable partial credit will be awarded and trying the problems will help you a lot for the exams.
- All problem numbers correspond to our text 'Introduction to Theory of Computation' by Boaz Barak. So, exercise a.b refers to Chapter a, exercise b.

1. If S, T are two finite sets, prove that there is a one-to-one mapping from S to T if and only if there is an onto mapping from T to S. [1 point]

   We have to show the if and the only if part.

   If part. Suppose that there is a one-to-one mapping from S to T. Let $E : S \to T$ be such a one-to-one mapping. Define a new mapping D from T to S as follows: for every $x \in S$, define $D(E(x)) = x$. This ensures that the mapping D covers all elements of S. However, there may be some other elements in T; we can assign an arbitrary value to these. Let x_0 be a fixed element in S. For all $y \in T \setminus \{E(x) : x \in S\}$ (i.e., for all elements y in T that are not in the image of S), set $D(y) = x_0$.

   Only if part. Suppose that there is an onto mapping from T to S. Let $D : T \to S$ be such an onto mapping. For every $x \in S$, let $I(x) = \{y \in T : D(y) = x\}$. As D is onto, $I(x)$ is non-empty for every x. We can define a one-to-one mapping E from S to T as follows: For all x, let $E(x)$ be a fixed element of $I(x)$. Then, E is clearly one-to-one.

2. Exercise 2.4. [1 point]

The idea is to use the approach we used in class for obtaining encodings of lists of objects from encodings of individual objects.

First, note that we can identify the set of vertices of the graph G with elements of $V = \{0, 1, 2, \ldots, n-1\}$. The edges is a set of pairs of the form (i, j) where $i, j \in V$. For $i = 0, \ldots, n-1$, let $N(i)$ denote the list of vertices that have an edge from i. Thus, the graph is described completely by the list $[N(0), N(1), \ldots, N(n-1)]$. Note that each $N(i)$ is itself a list of numbers from V. Further, by our assumption each list $N(i)$ has at most 10 entries (each vertex has at most 10 edges).

Therefore, to solve the problem it will suffice to have an encoding of lists of lists of the form $[N(0), N(1), \ldots, N(n-1)]$. We can do this by first encoding integers from V using binary encoding. As we did in class, we can obtain a prefix-free encoding $E_1 : V \to \{0,1\}^*$ where the length of the encoding of any integer i is at most $2\lceil \log i \rceil + 2 \leq 2\log n + 2$. Now, we can concatenate these prefix-free encodings to get an encoding function E_2 of the inner lists. Since each inner list has at most 10 vertices, the size of this encoding will be at most $10(2\log n + 2)$. Now, we have to build a representation for a list of such objects. To do this, we first convert E_2 to be prefix-free. Doing so will cause the encoding of the lists to increase to $2(20\log n + 20) + 2 = 40\log n + 42$. Finally, we can now concatenate the individual encodings to obtain the final encoding of the list of lists. The length of the final encoding will be $n(40\log n + 42) = 40n\log n + 42n < 1000n\log n$.

Some of you might have used the following alternate solution which may be a bit more involved than the above (given what we covered in class). We can count the number of graphs satisfying the constraints in the problem and argue that it is smaller than $2^{1000n\log n}$. If this is true, then there will be a one-to-one function as desired: If we have two finite sets S, T with $|S| \leq |T|$, then there is a one-to-one mapping from S to T. Now, to get an upper bound on the number of graphs, note that to specify the graph we have to specify the neighbors (the edges coming out) of each vertex. For each vertex, we have to specify its neighbors and we have at most n options for each of the 10 edges (if the vertex has fewer than 10 vertices, we can put a special symbol). So there are n^{10} options for what the neighbors of a vertex can be, and there are n vertices. So the total number of graphs is at most $(n^{10})^n = n^{10n} = 2^{10n\log n}$.

3. Exercise 3.3. [1 point]

   For any two bits $AND(a, b) = NOT(OR(NOT(a), NOT(b))$. So, we can replace every AND with this mini circuit.

4. Exercise 3.4. To be more precise, the problem is asking you to show that there is a function that **cannot** be computed by a Boolean circuit that is only allowed to use AND/OR (so NOT gates allowed). As a further hint, you can show that there is a function that takes two inputs and has one output that cannot be computed in such a way (no matter how many AND/OR gates you use). [1 point]

   Proof The function we will consider is $f : \{0,1\} \to \{0,1\}$ defined by $f(x) = \text{NOT}(x)$. To prove that no ANDOR-CIRC computes NOT, we need a definition and a sub-claim.

   Definition: We say that a function $f : \{0,1\}^n \to \{0,1\}$ is *monotone* if for every $x, y \in \{0,1\}^n$ if $x \leq y$ (i.e., $x_i \leq y_i$ for every $i \in [n]$) then $f(x) \leq f(y)$.

Note that NOT is not a monotone function. We will claim below that every function computed by an ANDOR-CIRC is monotone and so ANDOR-CIRC's can't compute NOT.

Sub-Claim: If f is computed by an ANDOR-CIRC then f is monotone.

Proof: Again this can be proved by induction on the size k of the ANDOR-CIRC. We won't do all the details, but summarize the key steps[1]. The base case with $k = 1$ is straightforward since the constant functions 0, 1, and the functions AND and OR are monotone. The (interesting) inductive step now considers a function computed as the AND or the OR of two previously computed functions. (A detailed proof would also ask you to check all the other cases, such as y_0 being a constant, or one of the two inputs being a variable, but these are the kind of details you can skip if you state that you are doing so explicitly — as we did by this parenthetical remark!) To see that functions computed as the AND or the OR of two previously computed functions are monotone it suffices to verify that if g and h are monotone, then so are AND$(g(x), h(x))$ and OR$(g(x), h(x))$. More generally if $f : \{0,1\}^m \to \{0,1\}$ and $g_1, \ldots, g_m : \{0,1\}^n \to \{0,1\}$ are monotone, then so is $F(x) = f(g_1(x), \ldots, g_m(x))$. To see this suppose $F(x) = 1$ and $F(y) = 0$. Then by monotonicity of f there must exist j such that $g_j(x) = 1$ and $g_j(y) = 0$. For if not, we would have $g_j(x) \leq g_j(y)$ for all j, which in turn would imply by monotonicity of f that $F(x) = f(g_1(x), \ldots, g_m(x)) \leq f(g_1(y), \ldots, g_m(y)) = F(y)$ a contradiction. In turn by monotonicity of g_j (and the same argument), there must exists i such that $x_i = 1$ and $y_i = 0$ and so $x \not\leq y$. This concludes the inductive step, and hence the Sub-Claim and hence the Claim. $\square$ $\square$

[1]In particular, the argument is complete when f is any function on a single bit as is enough to deal with the NOT function we are looking at.