3 Introduction to computer science

Exercises: 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, 3.16, 3.17, 3.18, 3.19, 3.20, 3.21, 3.22, 3.23, 3.24, 3.25, 3.26, 3.27, 3.28, 3.29, 3.30, 3.31, 3.32.

3.1

Since Turing machines can be thought of as computing functions from non-negative integers to non-negative integers (or in general, from reals to reals within finite precision), if a process in nature computes a function that consists of a map between different sets, it would not be computable by a Turing machine.

3.2

Considering a Turing machine with m possible internal states q_i, and an alphabet of n possible symbols Γ_j, any program line can be written as

$$\langle q_i, \Gamma_j, q_{i'}, \Gamma_{j'}, s \rangle .$$

Now, we map each program line to a string constructed in the following form:

- For each q_i we add the character "S" i times.

- For each Γ_j we add the character "A" j times.

- For $s = -1, 0$ or 1 we add, respectively, the characters "L", "K" or "R".

In the end, considering the program has N lines, it can be entirely mapped to the string

$$\underbrace{\mathrm{S}\cdots\mathrm{S}}_{i_1\text{ times}}\underbrace{\mathrm{A}\cdots\mathrm{A}}_{j_1\text{ times}}\underbrace{\mathrm{S}\cdots\mathrm{S}}_{i'_1\text{ times}}\underbrace{\mathrm{A}\cdots\mathrm{A}}_{j'_1\text{ times}}\underbrace{\mathrm{I}_1}_{\text{L, K or R}}\cdots\underbrace{\mathrm{S}\cdots\mathrm{S}}_{i_N\text{ times}}\underbrace{\mathrm{A}\cdots\mathrm{A}}_{j_N\text{ times}}\underbrace{\mathrm{S}\cdots\mathrm{S}}_{i'_N\text{ times}}\underbrace{\mathrm{A}\cdots\mathrm{A}}_{j'_N\text{ times}}\underbrace{\mathrm{I}_N}_{\text{L, K or R}} .$$

Now we make the character $\to$ number association

$$\mathrm{S}\to 0,\ \mathrm{A}\to 1,\ \mathrm{L}\to 2,\ \mathrm{K}\to 3,\ \mathrm{R}\to 4,$$

and for each character a_i of the string, we use its associated number a_i to calculate $p_i^{a_i}$, where p_i is the i-th prime number. With that, any Turing machine can be uniquely identified with the number

$$\prod_{i=1}^{l} p_i^{a_i},$$

where l denotes the length of the Turing machine's corresponding character string.

3.3

Consider a two-tape Turing machine where the input is written in tape 1 and tape 2 is initially blank. Then we can output, in tape 2, the reversed input number by using

$$\begin{aligned}
1 &: \langle q_s, \rhd, \rhd, q_1, \rhd, \rhd, +1, +1\rangle \\
2 &: \langle q_1, 0, b, q_1, 0, b, +1, 0\rangle \\
3 &: \langle q_1, 1, b, q_1, 1, b, +1, 0\rangle \\
4 &: \langle q_1, b, b, q_2, b, b, -1, 0\rangle \\
5 &: \langle q_2, 0, b, q_2, 0, 0, -1, +1\rangle \\
6 &: \langle q_2, 1, b, q_2, 1, 1, -1, +1\rangle \\
7 &: \langle q_2, \rhd, b, q_h, \rhd, b, 0, 0\rangle \,.
\end{aligned}$$

3.4

Consider a two-tape Turing machine where the input $x\,b\,y$ is written in tape 1 and tape 2 is initially blank. Then we can output, in tape 2, $x + y \mod 2$ by using

$$\begin{aligned}
1 &: \langle q_s, \rhd, \rhd, q_1, \rhd, \rhd, +1, +1\rangle \\
2 &: \langle q_1, 0, b, q_1, 0, 0, +1, 0\rangle \\
3 &: \langle q_1, 1, b, q_1, 1, 1, +1, 0\rangle \\
4 &: \langle q_1, 0, 0, q_1, 0, 0, +1, 0\rangle \\
5 &: \langle q_1, 1, 0, q_1, 1, 1, +1, 0\rangle \\
6 &: \langle q_1, 0, 1, q_1, 0, 0, +1, 0\rangle \\
7 &: \langle q_1, 1, 1, q_1, 1, 1, +1, 0\rangle \\
8 &: \langle q_1, b, 0, q_2, b, 0, +1, 0\rangle \\
9 &: \langle q_1, b, 1, q_2, b, 1, +1, 0\rangle \\
10 &: \langle q_2, 0, 0, q_2, 0, 0, +1, 0\rangle \\
11 &: \langle q_2, 1, 0, q_2, 1, 0, +1, 0\rangle \\
12 &: \langle q_2, 0, 1, q_2, 0, 1, +1, 0\rangle \\
13 &: \langle q_2, 1, 1, q_2, 1, 1, +1, 0\rangle \\
14 &: \langle q_2, b, 0, q_3, b, 0, -1, 0\rangle \\
15 &: \langle q_2, b, 1, q_3, b, 1, -1, 0\rangle \\
16 &: \langle q_3, 0, 0, q_h, 0, 0, 0, 0\rangle \\
17 &: \langle q_3, 1, 0, q_h, 1, 1, 0, 0\rangle \\
18 &: \langle q_3, 0, 1, q_h, 0, 1, 0, 0\rangle \\
19 &: \langle q_3, 1, 1, q_h, 1, 0, 0, 0\rangle \,.
\end{aligned}$$

3.5

Suppose there exists an algorithm $H_1(\cdot)$ that, upon receiving a Turing machine M as input, yields the result

$$H_1(M) = \begin{cases} 0 & ; \text{ if } M \text{ doesn't halt for blank input} \\ 1 & ; \text{ if } M \text{ does halt for blank input} \end{cases} .$$

If $H_1(\cdot)$ exists then we can define another algorithm H_2 that satisfies

$$H_2 \begin{cases} \text{halts} & ; \text{ if } H_1(H_2) = 0 \\ \text{never halts} & ; \text{ if } H_1(H_2) = 1 \end{cases} .$$

Now we ask the question: Does H_2 halt for blank input? There are two possible answers:

- H_2 does halt for blank input $\Rightarrow$ $H_1(H_2) = 1$ $\Rightarrow$ H_2 never halts.

- H_2 doesn't halt for blank input $\Rightarrow$ $H_1(H_2) = 0$ $\Rightarrow$ H_2 does halt.

In both cases we have a contradiction, so $H_1(\cdot)$ cannot exist meaning there are no algorithms capable of telling if M halts for blank input or not.

3.6

Suppose there exists an algorithm $H_p(x)$ that computes the probabilistic halting function correctly with probability strictly larger than 1/2. Then we can define an algorithm A that satisfies

$$A \begin{cases} \text{halts} & ; \text{ if } H_p(A) = 0 \\ \text{never halts} & ; \text{ if } H_p(A) = 1 \end{cases} .$$

Now we ask the question: Does A halt for input A with probability $\geq 1/2$? There are two possible scenarios:

- The answer is "yes":

  $h_p(A) = 1$ $\Rightarrow$ $H_p(A) = 1$ with probability $p > 1/2$ $\Rightarrow$ A never halts with probability $p > 1/2$ $\Rightarrow$ A halts with probability $p < 1/2$ $\Rightarrow$ $h_p(A) = 0$ $\Rightarrow$ the answer is "no".

- The answer is "no":

  $h_p(A) = 0$ $\Rightarrow$ $H_p(A) = 0$ with probability $p > 1/2$ $\Rightarrow$ A halts with probability $p > 1/2$ $\Rightarrow$ $h_p(A) = 1$ $\Rightarrow$ the answer is "yes".

In both cases we have a contradiction, so $H_p(x)$ cannot exist meaning there are no algorithms capable of correctly computing $h_p(x)$ with probability strictly larger than 1/2.

3.7

Yes, it is still undecidable, at least for computing the halting function for Turing machines augmented by the power of calling the oracle. The proof is analogous to the usual proof for the halting problem since one can always define a program for a Turing machine that halts if the oracle returns 0 for its own input and loops forever if the oracle returns 1. An oracle would only allow the computation of the halting function for Turing machines without oracle.

3.8

Denoting AND $:= \wedge$, NAND $:= \overline{\wedge}$, NOT $:= \neg$, OR $:= \vee$ and XOR $:= \oplus$ we have:

$$\begin{aligned} 1 \,\overline{\wedge}\, (a \,\overline{\wedge}\, b) &= 1 \,\overline{\wedge}\, \neg\,(a \wedge b) \\ &= \neg\,(1 \wedge \neg\,(a \wedge b)) \\ &= a \wedge b, \end{aligned}$$

$$\begin{aligned} (a \,\overline{\wedge}\, (1 \,\overline{\wedge}\, b)) \,\overline{\wedge}\, (b \,\overline{\wedge}\, (1 \,\overline{\wedge}\, a)) &= (a \,\overline{\wedge}\, \neg\,(1 \wedge b)) \,\overline{\wedge}\, (b \,\overline{\wedge}\, \neg\,(1 \wedge a)) \\ &= \neg\,(a \wedge \neg\,(1 \wedge b)) \,\overline{\wedge}\, \neg\,(b \wedge \neg\,(1 \wedge a)) \\ &= \neg\,(\neg\,(a \wedge \neg\,(1 \wedge b)) \wedge \neg\,(b \wedge \neg\,(1 \wedge a))) \\ &= \neg\,(\neg\,(a \wedge \neg b) \wedge \neg\,(b \wedge \neg a)) \\ &= (a \wedge \neg b) \vee (b \wedge \neg a) \\ &= a \oplus b, \end{aligned}$$

$$\begin{aligned} 1 \,\overline{\wedge}\, a &= \neg\,(1 \wedge a) \\ &= \neg a. \end{aligned}$$

3.9

If $f(n)$ is $O(g(n))$ then there are constants c and n_0 such that, for $n > n_0$, we have

$$f(n) \leq cg(n) \quad \Longleftrightarrow \quad \frac{1}{c} f(n) \leq g(n),$$

for constants $1/c$ and $n_0 \;\Rightarrow\; g(n)$ is $\Omega(f(n))$. The converse is immediate.

If $f(n)$ is $\Theta(g(n))$ then $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$, and using the previous result we have

$$\begin{aligned} f(n) \text{ is } O(g(n)) \quad &\Longleftrightarrow \quad g(n) \text{ is } \Omega(f(n)), \\ f(n) \text{ is } \Omega(g(n)) \quad &\Longleftrightarrow \quad g(n) \text{ is } O(f(n)). \end{aligned}$$

Thus $f(n)$ is $\Theta(g(n))$ if and only if $g(n)$ is $\Theta(f(n))$.

3.10

$$g(n) = \sum_{i=0}^{k} a_i n^i \quad \Longrightarrow \quad g(n) \text{ is } O\big(n^k\big),$$

so for $l \geq k$ we have that $g(n)$ is $O\big(n^l\big)$.

3.11

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}n}\log(n) &= n^{-1},\\ \frac{\mathrm{d}}{\mathrm{d}n}n^k &= kn^{k-1},\end{aligned}$$

and for $k > 0$, if we take $c = 1/k$, then for all $n > 1$ we have

$$n^{-1} \leq ckn^{k-1} \quad\implies\quad \log(n) \leq cn^k,$$

which means that $\log(n)$ is $O\big(n^k\big)$ for $k > 0$.

3.12

First let us analyse the case $k < 0$. In this case, $n^k = n^{\log n}$ for $n = 1$, and for $n > 1$ we have

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}n}n^k &= kn^{k-1} < 0,\\ \frac{\mathrm{d}}{\mathrm{d}n}n^{\log n} &= 2\log(n)n^{\log n - 1} > 0,\end{aligned}$$

so clearly n^k is $O\big(n^{\log n}\big)$ and $n^{\log n}$ cannot be $O\big(n^k\big)$ for $k < 0$. Now, for $k \geq 0$, $n^k = n^{\log n}$ for $n = e^k$, and for $n > e^k$, that is $n = ce^k$ for some constant $c > 1$, we have

$$\begin{aligned}\left(\frac{\mathrm{d}}{\mathrm{d}n}n^k\right)_{n=ce^k} &= kc^{k-1}e^{k(k-1)},\\ \left(\frac{\mathrm{d}}{\mathrm{d}n}n^{\log n}\right)_{n=ce^k} &= 2\left(\log c + k\right)c^{\log c + k - 1}e^{k(\log c + k - 1)}.\end{aligned}$$

The derivative of $n^{\log n}$ is clearly larger than the derivative of n^k since $\log c > 0$, so n^k is $O\big(n^{\log n}\big)$ and $n^{\log n}$ cannot be $O\big(n^k\big)$ for $k \geq 0$. Therefore n^k is $O\big(n^{\log n}\big)$ for any k but $n^{\log n}$ is never $O\big(n^k\big)$.

3.13

Let us start by taking the logarithm of both functions, that is $\log(c^n) = n\log c$ and $\log\big(n^{\log n}\big) = [\log n]^2$. The derivative of both functions yields

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}n}\left(n\log c\right) &= \log c,\\ \frac{\mathrm{d}}{\mathrm{d}n}\left(\log n\right)^2 &= \frac{2}{n}\log n.\end{aligned}$$

If $c > 1$ then clearly, for some large enough n_0, we have

$$\frac{2}{n}\log n \leq \log c$$

for $n > n_0$ since $\lim_{n\to\infty}[\log(n)/n] = 0$ and $\log c > 0$. Thus

$$[\log n]^2 \le n \log c \quad \iff \quad \log\left(n^{\log n}\right) \le \log(c^n) \quad \iff \quad n^{\log n} \le c^n$$

for large enough n, thus $n^{\log n}$ is $O(c^n)$ and c^n cannot be $O\left(n^{\log n}\right)$. From that, it follows that c^n is $\Omega\left(n^{\log n}\right)$ and $n^{\log n}$ cannot be $\Omega(c^n)$.

3.14

If $e(n)$ is $O(f(n))$ and $g(n)$ is $O(h(n))$ there are constants c_1, c_2 and n_0 such that, for $n > n_0$

$$e(n) \le c_1 f(n),$$
$$g(n) \le c_2 h(n).$$

Multiplying both inequalities yields

$$e(n)g(n) \le c_1 c_2 \left[f(n)h(n)\right],$$

which means $e(n)g(n)$ is $O(f(n)h(n))$.

3.15

Every time a compare-and-swap operation is applied, 2 among the $n!$ possible initial orderings will be correct, therefore after k applications we have, at most, 2^k possible initial orderings correctly ordered. So the total number N of required applications for all $n!$ initial orderings is such that

$$2^N = n! \quad \Longrightarrow \quad N = \log(n!).$$

Using Stirling's approximation, for sufficiently large n and some constant c we have

$$N = cn\log n + O(\log n) \quad \Longrightarrow \quad cn\log n \le N \quad \Longrightarrow \quad N \text{ is } \Omega(n \log n).$$

3.16

**From errata:* $2^n/\log n$ should be $2^n/n$.

Let $f_n(k)$ be a function that computes the number of functions on n inputs that uses, at most, k gates. There are $4 = 2^{2^1}$ possible Boolean functions on one single input, they are: $B_1(x) = 0, B_2(x) = x, B_3(x) = \neg x$ and $B_4(x) = 1$. By induction it is straightforward to conclude that there are 2^{2^n} Boolean functions on n inputs. Thus if all Boolean functions on n inputs can be computed using at most k gates then we must have $f_n(k) \ge 2^{2^n}$. So, if we show that

$$f_n\left(\frac{2^n}{n}\right) < 2^{2^n}$$

it would mean that there exist Boolean functions on n inputs that require more than $2^n/n$ logic gates to compute. We can estimate an upper bound for $f_n(k)$ with a counting argument. For that we suppose all circuits to be composed of binary gates, that is, gates on two bit inputs and one bit output (it is a reasonable supposition since it is possible to build a universal set of logic gates using only binary gates). There are $2^{2^2} = 16$ possible binary gates, and each one's two inputs can be the output of one of the other $k-1$ gates or one of the n inputs, and the output can be used as input by any of the other $k-1$ gates or be part of the final answer, so for the i-th gate we have roughly

$$\begin{aligned} 16k\binom{n+k-1}{2} &= 16k\frac{(n+k-1)!}{2!\,(n+k-3)!} \\ &= 8k\,(n+k-1)\,(n+k-2) \end{aligned}$$

possibilities. Since the label i can be anything from 1 to k we must elevate it to the k-th power, yielding a number of about

$$[8k\,(n+k-1)\,(n+k-2)]^k$$

possible circuits. But this is clearly an over-count since the labels i are arbitrary, so in order to compensate we divide it by $k!$ giving us a rough estimate

$$\begin{aligned} f_n(k) &\approx \frac{[8k\,(n+k-1)\,(n+k-2)]^k}{k!} \\ &\leq \frac{\left[8k\,(n+k)^2\right]^k}{k!}. \end{aligned}$$

Using Stirling's approximation we have that $k! \geq (k/e)^k$, thus

$$f_n(k) \leq (8e)^k\,(n+k)^{2k}\,.$$

We shall eventually consider $k = 2^n/n$ so it is safe to assume $n < k$ meaning

$$f_n(k) < (32e)^k\,k^{2k}.$$

Now properly setting $k = 2^n/n$ yields

$$\begin{aligned} f_n\left(\frac{2^n}{n}\right) &< (32e)^{2^n/n}\left(\frac{2^n}{n}\right)^{2\times 2^n/n} \\ &= \left(\frac{32e}{n^2}\right)^{2^n/n} 2^{2^n} 2^{2^n} \\ &< \left(\frac{64e}{n^2}\right)^{2^n} 2^{2^n}. \end{aligned}$$

Notice that for large enough n (in this estimation it would be for $n > 8\sqrt{e} \approx 13$) it is guaranteed that $f_n(2^n/n) < 2^{2^n}$, thus there exist Boolean functions on n inputs that require no less than $2^n/n$ logic gates to compute.

3.17

If the factoring decision problem is in **P** then given a number N we can answer, in polynomial time, if N has a non-trivial factor $p_1 \leq \sqrt{N}$. If the answer is "no" then we solved the problem of finding the prime factors of N in polynomial time (since in this case N is itself a prime). If the answer is "yes" we can then answer, in polynomial time, if N/p_1 has a non-trivial factor $p_2 \leq \sqrt{N/p_1}$, and repeat this process until we obtain "no" as answer. In this case, we will have obtained all prime factors $\{p_1, p_2, \cdots\}$ of N in polynomial time. Therefore, if the factoring decision problem is in **P** then the problem of finding the prime factors of a number is also in **P**.

Conversely, if we can obtain the prime factors $\{p_1, p_2, \cdots\}$ of N in polynomial time, then we could answer if N has a non-trivial factor less than some number in polynomial time, since we could just obtain the answers $\{p_1, p_2, \cdots\}$ in polynomial time. That means if the problem of finding the prime factors of a number is in **P** then the factoring decision problem is also in **P**.

3.18

Clearly $\mathbf{P} \subseteq \mathbf{NP}$. And also, if some $L \in \mathbf{P}$ then its complement $\overline{L}$ must also be in **P** since, if it is possible to obtain "yes" for L in a decision problem in polynomial time with a Turing machine M, it is possible to use another Turing machine M' that simulates M and outputs "yes" if M outputs "no" for L, meaning M' would output "yes" for $\overline{L}$ in polynomial time, so it is also true that $\mathbf{P} \subseteq \mathbf{coNP}$. With that, let us suppose that $\mathbf{P} = \mathbf{NP}$, then

- $\forall\, L \in \mathbf{NP}$:

  $L \in \mathbf{P}$, but since $\mathbf{P} \subseteq \mathbf{coNP} \;\Rightarrow\; L \in \mathbf{coNP}$.

- $\forall\, L \in \mathbf{coNP}$:

  $\overline{L} \in \mathbf{NP} \;\Rightarrow\; \overline{L} \in \mathbf{P} \;\Rightarrow\; L \in \mathbf{P} \;\Rightarrow\; L \in \mathbf{NP}$.

We proved that $\mathbf{P} = \mathbf{NP} \;\Rightarrow\; \mathbf{coNP} = \mathbf{NP}$, so the contrapositive must be true, that is, $\mathbf{coNP} \neq \mathbf{NP} \;\Rightarrow\; \mathbf{P} \neq \mathbf{NP}$.

3.19

In order to check if two vertices i and j are connected one must simply start at i and "walk" through the edges until j is reached. In the worst case scenario, one must pass through all the other vertices before reaching j meaning that cn steps are required at most, where c is a constant, thus REACHABILITY is a problem that can be solved in $O(n)$ steps.

In order to check if a graph is connected one must simply run the REACHABILITY problem for j varying from 1 to n with $j \neq i$. In the worst case scenario $cn\,(n-1)$ steps are required meaning this problem can be solved in $O(n^2)$ steps.

3.20

If a vertex is connected to two edges it can be entered and left once, so if it connected to an even number of n edges it can be visited $n/2$ times without the necessity of crossing any edges more than once. And if we start at a vertex connected to an even number of edges then we must also end the walk on it since every time this vertex is visited two of its connecting edges are used meaning there would be one last edge for us to end the walk on it. So if all vertices are connected to an even number of edges it is always possible to start at one point, visit every vertex crossing each edge only once, and then finish where we started, completing a cycle.

This automatically provides an algorithm for finding an Euler cycle: if all vertices are connected to an even number of edges one must simply start at any vertex and visit all the others, always using a different edge, up to the point where we have walked through all the edges once.

3.21

If L_1 is reducible to L_2 then there exists a Turing machine M that receives $x \in L_1$ and outputs $R(x) \in L_2$ in polynomial time, and if L_2 is reducible to L_3 then there is another Turing machine M' that receives $y \in L_2$ and outputs $S(y) \in L_3$ in polynomial time. So we can create a Turing machine M'' as the composite of M and M', that is, M'' receives $x \in L_1$, simulates M, use $R(x)$ as input for M', simulates M' and then outputs $S(R(x)) \in L_3$ in polynomial time, meaning L_1 is reducible to L_3.

3.22

All other languages of the same complexity class can be reduced to L. If L is reducible to L' then all other languages of the same complexity class can be reduced to L', meaning L' is complete.

3.23

-

3.24

-

3.25

A Turing machine, with l possible internal states and an alphabet of m possible symbols, that belongs in **PSPACE** uses $p(n)$ symbols of information, where n is the number of input symbols and $p(n) = \sum_i^k a_i n^i$. So, there is a total of $lm^{p(n)}$ possible states in which this Turing machine can be. We must assume that the machine halts for at least one of these states and that it doesn't enter in any infinite loop. So in the worst case scenario this machine must go through some multiple of all possible states before halting, meaning a number of steps given by

$$\begin{aligned} clm^{p(n)} &= cl \times 2^{p(n)\log m} \\ &= cl \times 2^{\log m \sum_i^k a_i n^i}, \end{aligned}$$

where c is a constant. Clearly $b := cl$ is a constant and all $\alpha_i := a_i \log m$ are also constants, so the running time of this machine is

$$b \times 2^{\sum_i^k \alpha_i n^i} = O\left(2^{n^k}\right) \quad \Longrightarrow \quad \mathbf{PSPACE} \subseteq \mathbf{EXP}.$$

3.26

Since $\log n \leq cn^k$ for constant $c \geq 1/k$ and $k > 0$ it means that

$$O(\log n) \leq O\left(n^k\right) \quad \Longrightarrow \quad \mathbf{L} \subseteq \mathbf{P}.$$

3.27

In the worst case scenario, a minimal vertex cover may be composed of just one of the two vertices at each step, in other words, it may be that in each step, among all edges touching either α or β, every single one touches α, meaning β would not need to be accounted. If this happens for every step then in the end VC is, at most, double the size of a minimal vertex cover.

3.28

It is obvious that $k \geq 3/4 \;\Rightarrow\; L \in \mathbf{BPP}$. For $1/2 < k < 3/4$ the probability of M correctly classifying x (accepting or rejecting it) is not high enough. But we can define a Turing machine M' that simulates M, running it n times for the same input x, and either accepts or rejects x based on the number of “yes” and “no” answers obtained by running M n times. Then the probability that M' will correctly classify x is the probability of having $n_c > n/2$, where n_c is the number of times M correctly classified x. This probability is given by

$$p = \sum_{n_c = \frac{n}{2}+1}^{n} \binom{n}{n_c} k^{n_c} (1-k)^{n-n_c},$$

so we would just need to find n such that $p \geq 3/4$, and that is always possible since $n \to \infty \;\Rightarrow\; n_c \to nk$, and because $k > 1/2$ we would have $n_c > n/2$ with probability $p \to 1$, thus $L \in \mathbf{BPP}$.

3.29

Denoting the Fredkin gate as F and considering the input state (a, b, c) we have

- If $c = 0$:

  $F(F(a, b, c)) = F(a, b, c) = (a, b, c).$

- If $c = 1$:

  $F(F(a, b, c)) = F(b, a, c) = (a, b, c).$

3.30

For $c = 0$: if both a and b are set to 1 they only collide in the center and close to the end, sending them to a' and b' respectively. If just one of them is set to 1 then they just follow their natural path to a' or b' respectively. And if both are set to 0 nothing happens obviously. So for $c = 0$ we have $F(a, b, 0)$ being correctly executed.

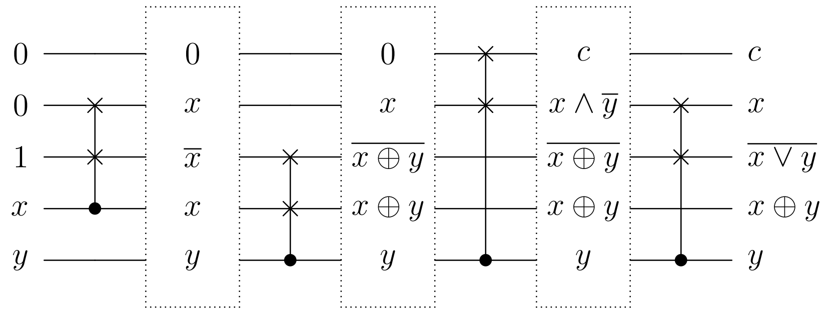
For $c = 1$: if both a and b are set to 1 there are 4 collisions between b and c and 5 collisions between a and b sending a, b and c to a', b' and c' respectively. If just one of them is set to 1 then they collide with c 4 times, ending up in b' or a' respectively, and sending c to c'. And if both are set to 0 c just follows its natural path to c'. So for $c = 1$ we have $F(a, b, 1)$ being correctly executed.

3.31

First, notice that $c = x \wedge y$, and

$$\begin{aligned}
&F(0,1,x) = (x, \neg x, x)\,,\\
&F(\neg x, x, y) = (\neg(x \oplus y), x \oplus y, y)\,,\\
&F(0, x, y) = (x \wedge y, x \wedge \neg y, y)\,,\\
&F(x \wedge \neg y, \neg(x \oplus y), y) = (x, \neg(x \vee y), y)\,.
\end{aligned}$$

Using these relations we can construct the following circuit, using three ancilla bits:



The boxes between each pair of Fredkin gates are just to indicate the state of each bit after each step. The bits at the end of the circuit can be outputted in the form $(x, y, c, x \oplus y)$ and the garbage bit, that is the one with value $\neg(x \vee y)$, can be discarded. Since we have only used Fredkin gates (and one NOT gate to set the third ancilla bit to 1 at the beginning) this circuit is reversible.

3.32

Simulating Toffoli with Fredkin:
Denoting the Toffoli gate as T, its action with control bits x and y and target bit t is given by $T(x, y, t) = (x, y, t \oplus (x \wedge y))$. We can output $t \oplus (x \wedge y)$ by using

$$F(\neg t, t, x \wedge y) = (\neg(t \oplus (x \wedge y)), t \oplus (x \wedge y), x \wedge y),$$

but first we need to create $\neg t, t$ and $x \wedge y$, and also need to generate the outputs x and y. To create $\neg t$ and t we can use two ancilla bits and apply

$$F(0,1,t) = (t, \neg t, t).$$

To create $x \wedge y$ and y we can use one ancilla bit and apply

$$F(0,x,y) = (x \wedge y, x \wedge \neg y, y),$$

but if we apply just it we won't be able to get output x since there will be no bits with value x, so first we need to use two more ancilla bits to create a copy of it with

$$F(0,1,x) = (x, \neg x, x).$$

So with five ancilla bits and four Fredkin gates it is possible to simulate a Toffoli gate.

Simulating Fredkin with Toffoli:
The action of the Fredkin gate with target bits a and b and control bit c is given by $F(a,b,c) = (a \oplus ((a \oplus b) \wedge c), b \oplus ((a \oplus b) \wedge c), c)$. We can output $a \oplus ((a \oplus b) \wedge c)$, $b \oplus ((a \oplus b) \wedge c)$ and c by applying

$$T(a \oplus b, c, a) = (a \oplus b, c, a \oplus ((a \oplus b) \wedge c)),$$
$$T(a \oplus b, c, b) = (a \oplus b, c, b \oplus ((a \oplus b) \wedge c)),$$

but first we need to create $a \oplus b$. For that we can use one ancilla bit and apply

$$T(b,1,a) = (b,1,a \oplus b),$$

but if we apply just it we won't be able to apply $T(a \oplus b, c, a)$ since there will be no bits with value a, so first we need to use one more ancilla bit to create a copy of it with

$$T(a,1,0) = (a,1,a).$$

So with two ancilla bits and four Toffoli gates it is possible to simulate a Fredkin gate.