Ph219/CS219 Problem Set 3

Solutions by Hui Khoon Ng

December 3, 2006

(KSV: Kitaev, Shen, Vyalyi, *Classical and Quantum Computation*)

3.1 The O(...) notation

We can do this problem easily just by knowing the following hierarchy:

$$s \leq (\log n)^p \leq n^q \leq r^n \tag{S1}$$

for any constants s, p, q, r greater than 1, and the log can be to any base. The "$\leq$" sign stands for the O-relation.

Let us transform some of the functions in question:

- $n \log n = cn \ln n$ (more precisely, $n \log_a n = n \frac{\ln n}{\ln a}$). Using the "=" sign for the bidirectional O-relation, we may write "$n \log n = n \ln n$". One must be careful, though – see the next item;

- $(\log_2 n)^n = (1/\ln 2)^n (\ln n)^n$. Note that $1/\ln 2 > 1$, hence $(\log_2 n)^n > (\ln n)^n$ in the O-relation sense;

- $(\ln n)^n = e^{(\ln \ln n)n}$. This function grows faster than $a^n = e^{(\ln a)n}$ for any constant a. Indeed $f(n) \leq O(g(n))$ implies that $e^{f(n)} \leq O\big(e^{g(n)}\big)$. Proof:

$$\begin{aligned} f \leq O(g) &\Rightarrow f(n) \leq cg(n) \qquad \forall n \geq n_0, \text{ some } c \\ &\Rightarrow e^{f(n)} \leq e^c e^{g(n)} \\ &\Rightarrow e^{f(n)} \leq (\text{constant}) e^{g(n)}; \end{aligned}$$

- $n^{\ln n} = e^{(\ln n)^2}$. This function grows faster than any constant power of n, yet not as fast as an exponential function.

- $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ using Stirling's formula;

- $\binom{2n}{n} = \frac{(2n)!}{(n!)^2} \approx \frac{4^n}{\sqrt{\pi n}}$;

- $\ln(n!) \approx \ln \sqrt{2\pi} + \frac{1}{2} \ln n + n \ln n \approx n \ln n$;

- $\ln\binom{2n}{n} \approx -\ln\sqrt{\pi} - \frac{1}{2}\ln n + n\ln 4 \approx n\ln 4$.

Putting all these together, the ordering of the functions is

$$n = \ln\binom{2n}{n} < n\log n = \ln(n!) < n^{\ln n} < 2^n < \binom{2n}{n} < (\ln n)^n < (\log_2 n)^n < n!\,.$$

3.2 Programming a Turing Machine

(a) We are given the n-digit binary number u written backwards on the tape as $\overline{x_0x_1x_2\dots x_{n-1}} = \sum_{k=0}^{n-1} 2^k x_k$, $x_k \in \{0,1\}$, and we want to increment it by 1. To do this, we start with the cell on the left end of the tape containing x_0, if it is 1, change it to 0, and move right by one step; if x_1 is 1, change it to 0 and move right by one step; etc. We repeat this until we see the first 0 or a blank symbol $\sqcup$ (marking the end of the input) on the tape, we change that to 1 and stop. This procedure is implemented by the rules below, with external alphabet given as $A = \{0,1\}$, internal alphabet $S = A \cup \{\sqcup\}$, and Turing machine (TM) states $Q = \{\alpha, \beta\}$, where α is the initial state of the TM and β will be used as the stop state. Recall the notation: $(q,s) \longrightarrow (q', t, \mathit{step})$ means the current TM state is q, the symbol in the current cell (where the head is) is s, and the transition is to a new state q', the head writes symbol t in current cell, and then moves by *step*. The rules are as follows:

$$\begin{aligned}(\alpha,0) &\longrightarrow (\beta,1,0)\\ (\alpha,1) &\longrightarrow (\alpha,0,+1)\\ (\alpha,\sqcup) &\longrightarrow (\beta,1,0).\end{aligned}$$

The TM stops when it goes into the β state. Note that this TM increments the input by 1 even if the input tape is blank (i.e. u is 0-bit number).

(b) We want to duplicate the input binary string bit by bit, i.e. we want to convert $x_0x_1\dots x_{n-1}$ into $x_0x_0x_1x_1\dots x_{n-1}x_{n-1}$. One way to do this is to clear the cell immediately to the right of the symbol we want to duplicate by shifting all symbols to the right by one cell. The external alphabet is given as $A = \{0,1\}$, we will use internal alphabet $S = A\bigcup\{\sqcup\}$, and the state set $Q = \{\alpha, r_0, r_1, l_0, l_1, w\}$. The procedure is as follows ($s, t \in \{0,1\}$):

Step 1: Pick up the symbol in the current cell which we want to duplicate, put a blank symbol in its place to mark the position, and move to the right. Swap the symbol we picked before with the one on the tape and move to the right again. Repeat until we see the end of the input.

$$\begin{aligned}(\alpha,s) &\longrightarrow (r_s,\sqcup,+1)\\ (r_s,t) &\longrightarrow (r_t,s,+1)\\ (r_s,\sqcup) &\longrightarrow (l_s,s,-1)\end{aligned}$$

Step 2: Move left until we find a blank symbol — the mark we left on Step 1. Put a copy of the symbol we saw last and make two step rightwards. Then return to Step 1.

$$
\begin{aligned}
(l_s, t) &\longrightarrow (l_t, t, -1) \\
(l_s, \llcorner\!\lrcorner) &\longrightarrow (w, s, +1) \\
(w, s) &\longrightarrow (\alpha, s, +1)
\end{aligned}
$$

Does this TM stop? After it is done duplicating the input, it will be in state α, but its current cell symbol will be $\llcorner\!\lrcorner$, for which there is no rule defined, so machine stops.

(c) The idea is to use a *fixed length* code based on the symbols in $A = \{0, 1\}$ to represent the internal alphabet $S \supset A$. Let us try to understand the details of how this works through an example. Suppose we have a TM that uses the internal alphabet $S = A \bigcup \{2, \llcorner\!\lrcorner\}$. Instead of using the additional symbol 2, we can have another TM that carries out the same program with the internal alphabet $S = A \bigcup \{\llcorner\!\lrcorner\}$ by using the following 2-bit code to represent all symbols of S:

$$0 \to 00, \qquad 1 \to 11, \qquad 2 \to 01, \qquad \llcorner\!\lrcorner \to 10.$$

We have chosen not to use the blank symbol in the encoding but rather reserve it for other purposes (see below).

When the input tape is given to the TM, with symbols from the external alphabet $A = \{0, 1\}$, the TM first does an encoding of the input, i.e., duplicating all input symbols according to the code so that the input $x_0 x_1 \ldots x_{n-1}$ becomes $x_0 x_0 x_1 x_1 \ldots x_{n-1} x_{n-1}$. Notice that this is exactly part (b) of this problem, the solution of which used only the internal alphabet consisting of $0, 1$, and $\llcorner\!\lrcorner$. Once the TM is done encoding the input, it performs the actual computation using the encoded input and the encoded internal symbols $2, \llcorner\!\lrcorner$. Note that the fixed length code allows the TM to recognize the correct internal symbol by reading the cells on the tape two at a time (one after another). When it is done, the TM returns to the left end of the tape and decodes the output back into 0's and 1's, i.e. $00 \to 0$ and $11 \to 1$.

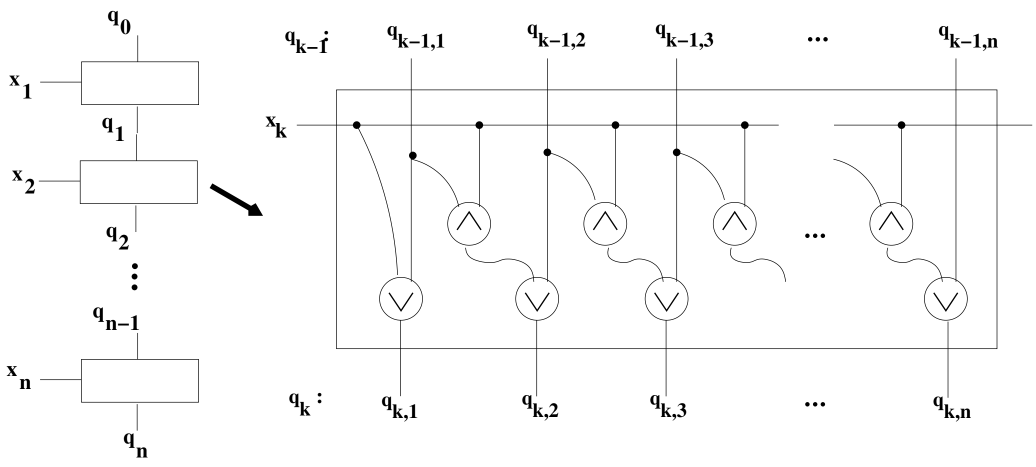
(There is a technical problem here: the definition given in class or in KSV does not allow a TM to see the left end of the tape before crashing into it. To cope with this difficulty, we may insert a blank symbol at the left end of the tape as a cushion. Furthermore, we have to avoid using blanks in our computation, which is the reason for the rule $\llcorner\!\lrcorner \to 10$. If the machine must move beyond the original encoded region, it will recognize a pair of blanks as an encoded blank, replacing it by the suitable combination of 0's and 1's. This way, 0's and 1's will always occupy a contiguous region, making it easy to locate the cushion symbol.)

A similar procedure can be used for a TM with S consisting of a larger set of symbols, just by using a longer code, with 0 represented as $00 \ldots 0$, 1 as $11 \ldots 1$, and other symbols represented by other codewords of the same length. The encoding phase can be carried out in a similar fashion as above, using only the internal symbols 0,1 and $\llcorner\!\lrcorner$.

3.3 The Threshold Function

(a) There are a few ways of doing this part, one of which is to think of this as sorting the input string of $x_1, \ldots, x_n$ into a string with all the 1's in front of all the 0's. Another way of doing this is to start with a string of n 0's, and shift 1's rightwards when the input is a 1. The circuit that does this is shown in Fig. 1, where $q_0 = 00\ldots0$ is a string of n 0's, and $q_k = q_{k,1}q_{k,2}\ldots q_{k,n}$ is the encoding of $y_k = \sum_{j=1}^{k} x_j$ as $\underbrace{1\ldots1}_{y_k}\underbrace{0\ldots0}_{n-y_k}$. In particular, q_n is the string representing $\sum_{j=1}^{n} x_j$ as required. The k-th bit of q_n gives the value of the THRESHOLD function.

Figure 1: The circuit for Problem 3 (a).

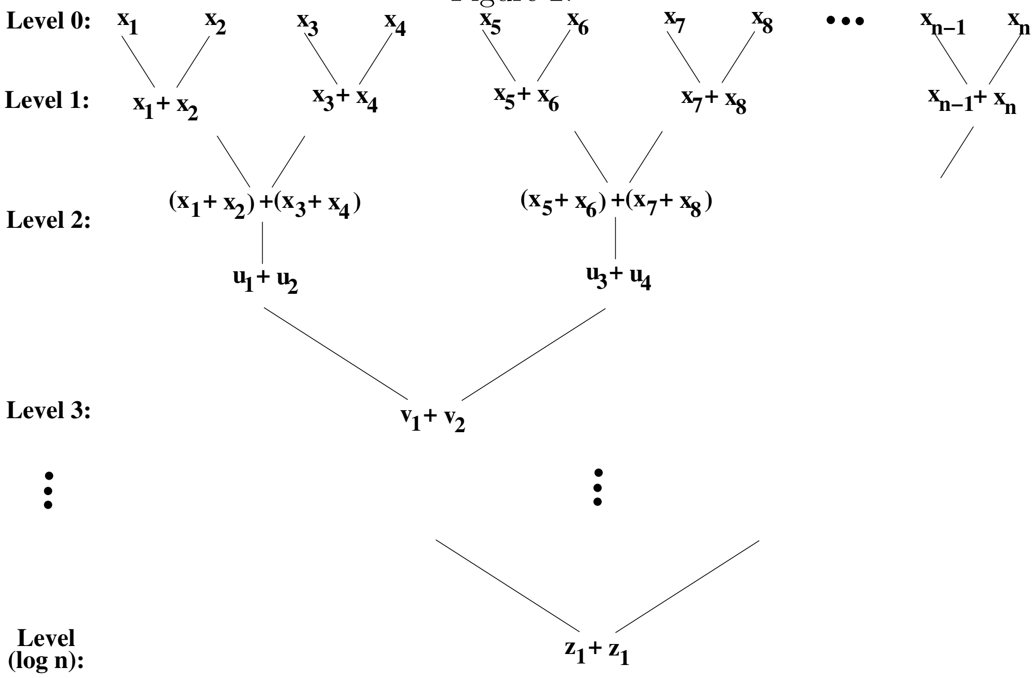


Each box in this circuit is of size $O(n)$ and constant depth, and we have n boxes in sequence. Therefore, the total circuit size is $O(n^2)$ and depth $O(n)$.

(b) Recall from class that there exists a circuit of size $O(m)$ and depth $O(1)$ that converts the sum of three m-digit numbers into a sum of two numbers, one of which is an m-digit number and the other an $(m+1)$-digit number. (If you want the details of the proof, refer to the lemma in the solution of KSV problem 2.13.) Using this circuit twice, we can convert a sum of four m-digit numbers into a sum of two $(m+1)$-digit numbers. We are going to use this in our algorithm described below.

Take $n = 2^l$ for some integer l (if this is not true, append the input data with 0's). Let us construct a tree, with the n bits of input data $x_1, x_2, \ldots, x_n$ as the leaves, and at each node, we add the values of the two input leaves to the node. At the first level, we do nothing, and just write $x_j + x_{j+1}$ at the node with input leaves x_j and x_{j+1}. At the second level, at each node, we convert the sum of four numbers $(x_{j-2} + x_{j-1}) + (x_j + x_{j+1})$ to a sum of two numbers using the constant depth circuit mentioned above. We end up with a sum of two 2-bit numbers at each node. Repeat this procedure for every level of the tree, each time converting the sum of four m-digit numbers at each node into a sum of two $m+1$-digit numbers. We end up with two $(\log n)$-digit numbers at the root of the tree. This tree is shown in Figure 2.

Figure 2:



Let us figure out the circuit complexity of this tree. At level 1, we do nothing, so the circuit size and depth are both constant. At level 2, we have $n/2$ nodes, at each of which, we convert four 1-bit numbers into two 2-bit numbers, so the size is $n/2 \cdot O(2)$, and depth is constant. At level k, we have $n/2^{k-1}$ nodes, at each of which, we convert four $(k-1)$-bit numbers into two k-bit numbers, so the size is $n/2^{k-1} \cdot O(k)$ and depth is still constant. We have $\log n$ levels in total, hence,

$$\text{size} = \sum_{k=1}^{\log n} \frac{nO(k)}{2^{k-1}} = O\left(n \sum_{k=0}^{\log n - 1} \frac{k}{2^k}\right) \leq O\left(n \sum_{k=0}^{\infty} \frac{k}{2^k}\right) = O(n),$$

since the series $\sum_{k=0}^{\infty} \frac{k}{2^k}$ converges. The depth is $O(\log n)$.

At the root of the tree, we have two $(\log n)$-digit numbers denoted as z_1 and z_2. We want a binary output at the end of the procedure, so we still have to do the actual binary sum of z_1 and z_2, which can be done with the parallel addition circuit with depth $O(\log\log n)$ and size $O(\log n)$.

Altogether then, the circuit complexity is:

$$\begin{aligned} \text{size} &= O(n) + O(\log n) = O(n), && \text{— better than the problem asks for!} \\ \text{depth} &= O(\log n) + O(\log\log n) = O(\log n). \end{aligned}$$

To get the value of the THRESHOLD function, we need to compute the difference $\sum_{j=1}^n x_j - k$ and compare the result with 0. To do this, we write the difference as $\sum_{j=1}^n x_j + (-k)$ and use the complementary code to represent $-k$ in binary. (Google "two's complement" if you are not familiar with this. Briefly, to represent negative numbers for m-digit numbers, we use an $(m+1)$-digit binary string, with the first bit being the sign bit, i.e. it is 0 if the number is non-negative, 1 if negative. To read the complementary code when the sign bit is 1, we flip all the bits in the $(m+1)$-bit string and add 1. The number represented is then the negative of the resulting string.) Then, do the binary sum of $\sum_{j=1}^n x_j$ and $-k$, which we can do with size $O(\log n)$ and depth $O(\log\log n)$ since both are $O(\log n)$-digit numbers, and the value of the THRESHOLD function is the value of the sign bit of the result.

3.4 Measurement Decoding

We are given n readings $s_0, s_1, \ldots, s_{n-1} \in \{0, 1, \ldots, 7\}$ from n meters differing in rotation speeds by a factor of 2, with the s_0 meter being the slowest. We are promised that these readings are consistent estimates of a number x between 0 and 1 such that $2^j x (\text{mod } 1) \in \left(\frac{s_j - 1}{8}, \frac{s_j+1}{8}\right)$. Let us define $\beta_j := \frac{s_j}{8}$, then,

$$\left|(2^j x - \beta_j)_{\text{mod } 1}\right| < \frac{1}{8} \qquad \text{for } j = 0, 1, \ldots, n-1.$$

The problem is to combine the overlapping information from the n readings to get an estimate y of x such that $|(x-y)_{\text{mod } 1}| < 2^{-(n+2)}$.

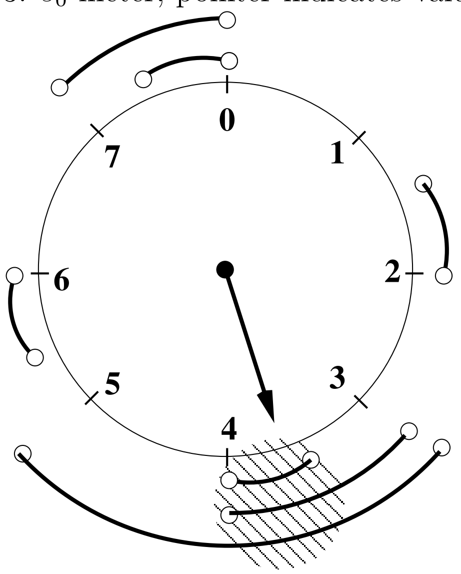
First, a simple fact that we will use: suppose $|(2a - \overline{.b_1 b_2 b_3 \ldots})_{\text{mod } 1}| < \delta < \frac{1}{2}$, then exactly one of the following is true:

$$\begin{aligned} \text{either} \qquad & \left|\left(a - \overline{.0b_1b_2\ldots}\right)_{\text{mod } 1}\right| < \delta/2 \\ \text{or} \qquad & \left|\left(a - \overline{.1b_1b_2\ldots}\right)_{\text{mod } 1}\right| < \delta/2. \end{aligned} \tag{S2}$$

For each j, let us write $\beta_j := \overline{.b_{j1}b_{j2}b_{j3}}$ (note that we only need three digits since $\beta_j = \text{integer}/8$). Given $|(2^j x - \beta_j)_{\text{mod } 1}| < 1/8$, using (S2), we also know that either $|(2^{j-1}x - \overline{.0b_{j1}b_{j2}b_{j3}})_{\text{mod } 1}| < 1/16$ or $|(2^{j-1}x - \overline{.1b_{j1}b_{j2}b_{j3}})_{\text{mod } 1}| < 1/16$. Repeating this procedure, we eventually get 2^j possible regions, each of length $2^{-(j+2)}$, in which x can lie which are consistent with the given value of β_j. Since the data are consistent, there will be a finite region contained in the intersection of the sets of regions from the different j values. This region is of length $2^{-(n+1)}$ and contains x. If we take y as the midpoint of this region, then $|(x-y)_{\text{mod } 1}| < 2^{-(n+2)}$.

Let us look at a specific example. Consider $x = \overline{.011101}$, which we know just to check our results. Take $n = 3$. Then, we can have the data $s_0 = 4, s_1 = 7, s_2 = 7$. We can check that this

Figure 3: s_0 meter, pointer indicates value of x.



set of data satisfies the required conditions:

$$\left|\left(x-\frac{s_0}{8}\right)_{\text{mod }1}\right| = \left|\overline{.011101}-\overline{.100}\right| = \left|-\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{64}\right| = \frac{3}{64} < \frac{1}{8},$$
$$\left|\left(2x-\frac{s_1}{8}\right)_{\text{mod }1}\right| = \left|\overline{.11101}-\overline{.111}\right| = \frac{1}{64} < \frac{1}{8},$$
$$\left|\left(2^2x-\frac{s_2}{8}\right)_{\text{mod }1}\right| = \left|\overline{.1101}-\overline{.111}\right| = \left|-\frac{1}{8}+\frac{1}{16}\right| = \frac{1}{16} < \frac{1}{8}.$$

Using (S2) repeatedly, from s_2, we get four possible regions in which x can lie, each of length 1/16 (see Fig. 3), from s_1, two regions, each of length 1/8, and from s_0, one region of length 1/4. The shaded region (of length 1/16 as constrained by the region coming from s_2) corresponds to the overlap between all the different s_j values. As can be seen from the diagram, x indeed lies in this region, and we get the desired precision if we take y as the midpoint of the shaded region.

(a) How do we get the correct value of y with circuit size $O(n)$? The problem is, of course, that there are exponentially many regions. Fortunately, we do not need to list all of them.[1] Let

[1] A subtle point to note is that there is more than one possible set of values for $\{s_j\}$ that is consistent with a given x. For instance, in the example above, s_0 could have been 3 instead of 4, and we would still get the same shaded region. This causes the following seemingly plausible strategy to *fail* to give the correct y: write each β_j in the binary fraction representation $\beta_j = \overline{.b_{j1}b_{j2}b_{j3}}$ and take $y = \overline{.b_{01}b_{11}b_{21}\ldots b_{n-2,1}b_{n-1,1}b_{n-1,2}b_{n-1,3}}$, i.e. start with $\beta_{n-1} = \overline{.b_{n-1,1}b_{n-1,2}b_{n-1,3}}$, and prepend with the first digit of $\beta_{n-2}, \beta_{n-3}, \ldots, \beta_0$. In our example, if the data is given as $s_0 = 3, s_1 = 7, s_2 = 7$, this strategy gives $y = \overline{.01111}$ which is within 1/16 of the value of x. However, if the data is given as $s_0 = 4, s_1 = 7, s_2 = 7$, we get $y = \overline{.11111}$, which is *not* within the desired precision. It is easy to see that this strategy only works if we are given s_j values that are the nearest tick marks counterclockwise from the pointer for $2^j x$ on the respective meters.

us go back to our original argument. The inequalities $|(2^{j-1}x - \overline{.0b_{j1}b_{j2}b_{j3}})_{\text{mod }1}| < 1/16$ and $|(2^{j-1}x - \overline{.1b_{j1}b_{j2}b_{j3}})_{\text{mod }1}| < 1/16$ define two arcs of length 1/8 on opposite sides of the circle. Only one of the arcs can overlap with the region defined by the inequality $|(2^{j-1}x - \beta_{j-1})_{\text{mod }1}| < 1/8$ because the length of this region (1/4) is smaller than the gaps between the arcs (3/8). Therefore, we can select the right arc before further multiplying the number of choices. This observation suggest the following algorithm, in which we work out the bits of y *right to left*.

Let us start with the binary representation of $\beta_{n-1} := \overline{.y_n y_{n+1} y_{n+2}}$. Now, sequentially pick y_k, $k = n-1, n-2, \ldots, 1$ such that

$$y_k = \begin{cases} 0 & \text{if } \left(\overline{.0y_{k+1}} - \beta_{k-1}\right)_{\text{mod }1} \in \left\{-\frac{2}{8}, -\frac{1}{8}, \frac{0}{8}, \frac{1}{8}\right\}, \\ 1 & \text{if } \left(\overline{.1y_{k+1}} - \beta_{k-1}\right)_{\text{mod }1} \in \left\{-\frac{2}{8}, -\frac{1}{8}, \frac{0}{8}, \frac{1}{8}\right\}. \end{cases} \tag{S3}$$

(Note that exactly one of these two conditions holds.) Then, take $y = \overline{.y_1 y_2 \ldots y_n y_{n+1} y_{n+2}}$. We need to check that y found this way gives the required precision for x, i.e., $|(x-y)_{\text{mod }1}| < 2^{-(n+2)}$. We prove this using induction on the statement:

$$\left|\left(\overline{.y_k y_{k+1} \ldots y_{n+2}} - 2^{k-1}x\right)_{\text{mod }1}\right| < \delta_k, \qquad \text{where } \delta_k = 2^{-(n+3-k)}. \tag{S4}$$

Base case. If $k = n$ then

$$\left|\left(\overline{.y_n y_{n+1} y_{n+2}} - 2^{n-1}x\right)_{\text{mod }1}\right| = |(\beta_{n-1} - 2^{n-1}x)_{\text{mod }1}| < 2^{-3} = \delta_n.$$

Induction step. Assume that equation (S4) is true for some k; we need to prove it for $\tilde{k} = k-1$. The inductive assumption implies that

$$\left|\left(\overline{.uy_k y_{k+1} \ldots y_{n+2}} - 2^{k-2}x\right)_{\text{mod }1}\right| < \frac{\delta_k}{2} = \delta_{k-1} \tag{S5}$$

for $u = 0$ or $u = 1$. We just need to check that (S3) gives the correct value for y_{k-1}, i.e., $y_{k-1} = u$. To see this, notice that

$$0 \le \overline{.uy_k y_{k+1} \ldots y_{n+2}} - \overline{.uy_k} \le \sum_{j=3}^{n+4-k} 2^{-j} = 1/4 - \delta_{k-1}. \tag{S6}$$

Using (S5) and (S6) together with the data consistency condition, $|(2^{k-2}x - \beta_{k-2})_{\text{mod }1}| \le 1/8$, we get:

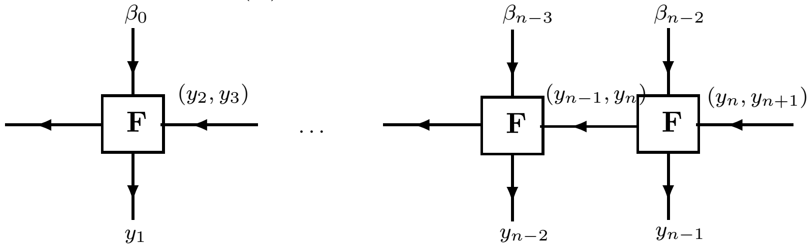
$$-\delta_{k-1} - (1/4 - \delta_{k-1}) - 1/8 \ < \ (\overline{.uy_k} - \beta_{k-2})_{\text{mod }1} \ < \ \delta_{k-1} - 0 + 1/8.$$

Hence

$$(\overline{.uy_k} - \beta_{k-2})_{\text{mod }1} \in \{-2/8, -1/8, 0, 1/8\} \tag{S7}$$

as required.

Figure 4: $O(n)$ circuit as a finite-state automaton.



In this algorithm, y_k depends only on β_k and y_{k+1}. Proving the correctness by induction is somewhat tricky, though. The proof may be easier to carry out if we replace condition (S3) by, say, this inequality:

$$\left|\left(\overline{.y_k y_{k+1} y_{k+2} 1} - \beta_{k-1}\right)_{\text{mod } 1}\right| < 1/4. \tag{S8}$$

In this case, y_k depends on β_k, y_{k+1}, and y_{k+2}. However, the overall structure of the algorithm remains the same. It can be viewed as a finite-state automaton $F : (\beta, q) \mapsto (y, q)$ (where $\beta \in \{0,1\}$, $y \in \{0,1\}$, and $q = (y', y'') \in \{0,1\}^2$ denotes the state of the automaton) defined as:

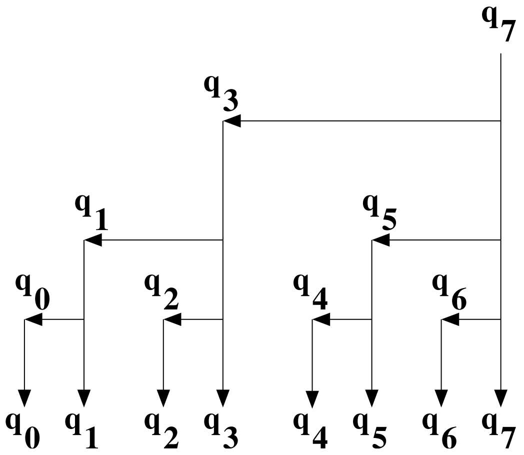
$$F(\beta_{k-1}, (y_{k+1}, y_{k+2})) = (y_k, (y_k, y_{k+1}))$$

with y_k given by (S8). This is represented schematically in Fig. 4. The initial state of the automaton is computed from β_{n-1} (which can be done with a constant size circuit). Notice that F is computable by a fixed size circuit, but we need to repeat it n times in sequence, so the circuit complexity is given by

$$\begin{aligned} \text{size} &= O(n) \\ \text{depth} &= O(n). \end{aligned}$$

(b) From part (a), we know that our algorithm can be viewed as a finite-state automaton operating n times in sequence. This point of view makes it very easy to see how to parallelize the circuit, just by analogy with the parallelization of addition discussed in class. Recall that in parallelizing addition, we compute all the carry-bits with depth $O(\log n)$ by first computing the composition of functions given the input in the form of a tree of depth $O(\log n)$. Here, the automaton state $q_k := (y_{k+1}, y_{k+2})$ plays the role of the carry bit, which we need to compute the next value of y_k. We carry out the exact same procedure as in the case of parallelizing addition. We first compute all compositions of the functions $F_{\beta_k} : Q \longrightarrow Q$ in the form of a tree. This tree is of depth $O(\log n)$, with $O(n)$ nodes, each node corresponding to a composition of two fixed size functions (and hence computable by a circuit with constant size and depth). Hence this composition process has circuit size $O(n)$ and depth $O(\log n)$. Having gotten all these composition functions, we need to compute $q_k \forall k$. This can be done in $O(\log n)$ steps, just like in the

Figure 5: Computing q values for n=8.



case of computing the carry bits in the parallel addition algorithm (see Figure 5 for an example for $n = 8$). Finally, we compute in parallel, all the y_k values given the q_k values. Altogether then, the circuit complexity is:

$$\text{size} = O(n),$$
$$\text{depth} = O(\log n).$$

Note that all finite-state automaton algorithms of the above form can be parallelized in the same way to reduce the depth from $O(n)$ to $O(\log n)$. For further reference, see the solution to KSV problem 2.11 (for simplicity, you can take the parameter k in that problem to be a constant).