Queue Automata
Term 2 AY 2024 - 2025

Evan Mari De Guzman
2401 Taft Avenue
Manila, Philippines
evan_mari_deguzman@dlsu.edu.ph

Denise Liana Ho
2401 Taft Avenue
Manila, Philippines
denise_liana_ho@dlsu.edu.ph

ABSTRACT

The computational model Queue Automaton (QA) is a nondeterministic finite-state machine with an infinite-memory queue as auxiliary storage. It consists of finite sets of states, input alphabet, and queue symbols, along with an initial state, accepting states, an initial queue symbol, and a transition function based on input and operations (dequeue and enqueue) done on the queue. Queue Automata is similar to Pushdown Automata (PDA), except it follows a *first-in, first-out* policy when storing symbols in its memory. The machine processes an input string by transitioning through states while manipulating the queue, accepting if it reaches a final state with an empty queue. A Queue Automaton recognizes Recursively enumerable languages (Type-0 in the Chomsky hierarchy), making it Turing-equivalent. Queue Automata can be applied to bioinformatics, specifically in modeling biological computers that utilize DNA and proteins for computation. Sakowski et al. (2022) propose that it effectively schedules controlled genome sequence reading at the molecular level, with IIB restriction endonuclease facilitating DNA fragment transitions within the machine framework.

MACHINE DEFINITION

A Queue Automata, formally, is a machine capable of performing operations more complex than Finite-State Machines (FSM). This is due to its ability to hold a queue. With the queue, the machines under this definition are not bounded by states and are capable of storing and recalling infinite information. In the same manner as a Pushdown Automata, a QA is non-deterministic.

According to Jakobi et al. (2021), to mathematically define a QA, it is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ where:

- Q is finite set of states
- Σ is finite set of input symbols
- Γ is finite set of queue symbols
- $\delta : Q \times \Sigma_\lambda \times \Gamma_\lambda \to P(Q \times \Gamma_\lambda)$ is the transition function
- q_0 is the start state, where $q_0 \in Q$
- Z is the initial queue symbol, where $Z \in \Gamma$
- F is the set of accepting states, where $F \subseteq Q$

A QA consists of four components: (1) the finite control of input, (2) the set of input alphabet, (3) the transitions from state to state, and lastly, the most important, (4) a queue system integrated within the automata. In summation, the total state of a QA is the state of the machine and the queue line.

QA, as defined, is an accepter machine. It takes an input string $\omega \in \Sigma^*$. The input string will go through transitions defined as $\delta(q, s, \gamma) = (q', \gamma')$ where $q, q' \in Q, s \in \Sigma, \gamma \in \Gamma$. We create these transition functions until we get to the transition $\delta(q, s, Z) = (q_f, \lambda)$, where $q \in Q, q_f \in F, s \in \Sigma$. The output simply accepts or rejects a string ω when the machine transitions into a final state q_f and the queue is empty.

Transitions in QA are represented in state diagrams as $input; dequeue; enqueue$, where input comes from Σ_λ, and both dequeue and enqueue symbols come from Γ_λ Formally, a dequeue operation, $\delta(q, s, \gamma) = (q', \lambda)$, takes out the symbol γ if it is at the head of the queue and shifts the contents forward so that the symbol behind the original head becomes the new head. An enqueue operation, $\delta(q, s, \lambda) = (q', \gamma)$, adds the new symbol γ to the very end or tail of the queue.

The QA machine bears similarities in operation with a PDA. The only true difference between them is that instead of employing a *last in, first out* (LIFO) policy, a *first in, first out* (FIFO) policy is utilized. In summation, besides reading input symbols, a QA has an operation for reading and writing into a queue line.

Note that in PDA, the initial stack symbol Z is explicitly pushed at the start of any computation. In QA however, the initial queue symbol must be enqueued before the machine intends to empty the queue. This is due to its FIFO nature.

To complete this subsection of machine definition, the paper will provide an example of a QA machine given the language:

$$L = \{\omega \in \{0,1\}^* | \omega = 0^n 1^m, 1 \leq n \leq m\}$$

For this problem, the state diagram for a QA that recognizes L is provided in the figure below.

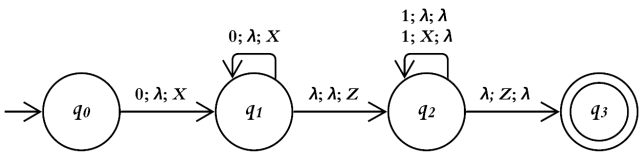


Figure 1. State diagram of a QA accepter that recognizes $L = \{\omega \in \{0,1\}^* | \omega = 0^n 1^m, 1 \leq n \leq m\}$

Given the input string $\omega = 00111$, which is a valid and accepted string within the machine ($1 \leq n \leq m$ holds for $n = 2, m = 3$), the movement of the machine can be traced as follows:

State	ω	Read	Dequeue	Enqueue	Queue	Next
q_0	$\underline{\mathbf{0}}0111$	0	λ	X	X	q_1
q_1	$0\underline{\mathbf{0}}111$	0	λ	X	XX	q_1
q_1	$0\underline{\mathbf{0}}111$	λ	λ	Z	XXZ	q_2
q_2	$00\underline{\mathbf{1}}11$	1	X	λ	XZ	q_2
q_2	$001\underline{\mathbf{1}}1$	λ	X	λ	Z	q_2
q_2	$0011\underline{\mathbf{1}}$	λ	λ	λ	Z	q_2
q_2	00111	λ	Z	λ	λ	q_3

Note: The bold, underlined symbol in ω represents the current symbol being read in ω. The queue *head* is on the left end, while the *tail* is on the right end.

Table 1. Trace Table a QA accepter that recognizes $L = \{\omega \in \{0,1\}^* | \omega = 0^n 1^m, 1 \leq n \leq m\}$

FORMAL LANGUAGE & COMPUTATIONAL POWER

Building on the previous comparison between QA and PDA, the computational power of a QA may be at least equal to that of a PDA, as both models are like FSMs but with auxiliary memory structures, to put it informally. We can position it within the Chomsky hierarchy for a more substantive approach to defining the power of a QA.

The Chomsky hierarchy categorizes formal languages into four levels based on their generative power. Each level corresponds to a class of formal languages recognized by a computational model, which allows us to define its theoretical power. An automaton that recognizes a broader set of languages is considered more powerful (EITCA, 2024).

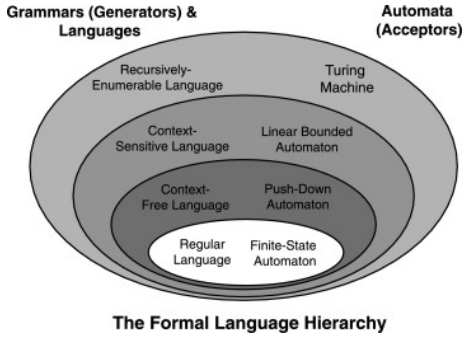


Figure 2. Visualization of the Chomsky Hierarchy. Adapted from Fitch (2014)

As shown in Figure 2, FSMs, which recognize Regular languages (Type-3) possess the lowest computational power, followed by those for Context-free (Type-2), and Context-sensitive (Type-1). Recursively Enumerable languages (Type-0) are recognized by the most powerful model (Turing Machines), as it can simulate any algorithm.

The power of QA in relation to FSM

We first prove that QA can recognize Regular languages and is strictly more powerful than FSM (either deterministic or nondeterministic) since FSMs lack access to an auxiliary memory structure. We begin by showing that a QA can simulate any given FSM.

A nondeterministic finite automaton (NFA), following Hopcroft et al. (2001), is the 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of states
- Σ is a finite set of input symbols
- $\delta : Q \times \Sigma \to P(Q)$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is a set of accepting states

From any given NFA $M_a = (Q_a, \Sigma_a, \delta_a, q_{0_a}, F_a)$, we can construct an equivalent QA $M_b = (Q_b, \Sigma_b, \Gamma, \delta_b, q_{0_b}, Z_0, F_b)$ as follows:

- $Q_b = Q_a$ (states remain unchanged)
- $\Sigma_b = \Sigma_a$ (input symbols remain unchanged)
- $\Gamma = \{Z_0\}$ (queue only contains initial symbol)
- $\forall q \in Q_a, \forall s \in \Sigma_a, \forall q' \in \delta_a(q,s), \quad \delta_b(q,s,\lambda) = (q',\lambda)$ (copy each transition without touching the queue)
- $q_{0_b} = q_{0_a}$ (start state remains unchanged)
- $Z_0 = Z$ (initial queue symbol remains)
- $F_b = F_a$ (final states remain unchanged)

Since the queue is never used in M_b, its behavior is identical to that of M_a. Formally, for any input string $\omega \in \Sigma_a^*$, the transitions of M_b mirror those of M_a, so M_b accepts ω if and only if M_a does.

Therefore, the class of languages recognized by an NFA is a subset of the set recognized by a QA, or $L(NFA) \subseteq L(QA)$. Since NFAs are equivalent to DFAs, and DFAs are equivalent to Regular expressions (Hopcroft et al., 2001), it follows that QAs can recognize all Regular languages by transitivity.

Now, consider the non-Regular language (proven with Pumping Lemma; Critchlow and Eck, 2011):

$$L = \{\omega \in \{a,b\} | \omega = a^n b^n, n \geq 0\}$$

NFA cannot recognize this language as they only recognize Regular languages, but the QA defined in Figure 3 can.

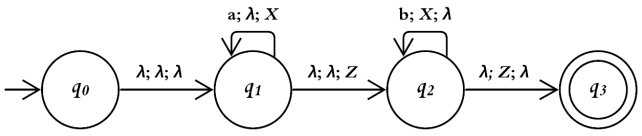


Figure 3. State diagram for the QA that recognizes $L = \{\omega \in \{a,b\} | \omega = a^n b^n, n \geq 0\}$

This shows that QAs can recognize languages other than Regular languages, whereas NFAs cannot. Therefore, the descriptive power of QAs is strictly more than NFAs, which means that $L(FSM) \subset L(QA)$.

The power of QA in relation to PDA

We now prove that QA can recognize Context-free languages and is strictly more powerful than PDA. Recalling from previous sections, the only difference between QA and PDA is that they employ different data structures, queue and stack, respectively.

According to Hopcroft et al. (2001), we can formally define a PDA as the 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where:

- Q is a finite set of states
- Σ is a finite set of input symbols
- Γ is a finite set of stack alphabet
- $\delta : Q \times \Sigma_\lambda \times \Gamma_\lambda \mapsto P(Q \times \Gamma_\lambda)$ is the transition function
- $q_0 \in Q$ is the initial state
- $Z_0 = Z$, is the initial stack symbol, $Z_0 \in \Gamma$
- $F \subseteq Q$ is a set of accepting states

For any given PDA $M_a = (Q_a, \Sigma_a, \Gamma_a, \delta_a, q_{0_a}, Z_a, F_b)$, we can construct an equivalent QA $M_b = (Q_b, \Sigma_b, \Gamma_b, \delta_b, q_{0_b}, Z_b, F_b)$ as follows:

- $Q_b = Q_a \cup Q_{rot}$ (Q_{rot} is a set of new states used to "pop")
- $\Sigma_b = \Sigma_a$ (input symbols remain unchanged)
- $\Gamma_b = \Gamma_a \cup \{\#\}$ (stack symbols and a special marker)
- $q_{0_b} = q_{0_a}$ (initial state remains unchanged)
- $Z_b = Z_a$ (initial stack symbol remains unchanged)
- $F_b = F_a$ (final states remain unchanged)
- $\forall q \in Q_a, \forall s \in \Sigma_a, \forall \gamma \in \Gamma_a, \forall (q', \gamma') \in \delta_a(q, s, \gamma)$,
  $\delta_b(q, s, \gamma) = \begin{cases} (q', \gamma') & \text{if } \gamma = \lambda \text{ (push} \rightarrow \text{enqueue)} \\ (q_{\text{rot}}, \#) & \text{if } \gamma \neq \lambda \text{ (pop} \rightarrow \text{queue rotation)} \end{cases}$
- **"Popping" in a Queue:** Whenever M_a performs a pop, M_b must access the queue the same way a stack would and "pop" from the tail. This is achieved by rotating the queue (Jan, 2012) with the following steps:
  1. $\delta_b(q_{\text{rot}}, \lambda, \lambda) = (q_{\text{rot1}}, \#)$ (Enqueue a special marker #)
  2. $\delta_b(q_{\text{rot1}}, \lambda, \gamma) = (q_{\text{rot1}}, \gamma)$ (Dequeue and re-enqueue each symbol in queue)
  3. $\delta_b(q_{\text{rot1}}, \lambda, \gamma) = (q_{\text{rot2}}, \lambda)$ (Dequeue the queue head, originally the tail before rotation)
  4. $\delta_b(q_{\text{rot2}}, \lambda, \#) = (q, \lambda)$ (Dequeue #. Pop completed!)

For any input string $\omega \in \Sigma_a$, M_b will only accept it if and only if it reaches a final state in M_a. Therefore, any language recognized by a PDA is also recognized by a QA, implying that L(PDA) $\subseteq$ L(QA). Since PDAs are equivalent to Context-free grammars (Hopcroft et al., 2001), it follows by transitivity that QAs can recognize Context-free languages.

Now, consider the non-Context-Free language (proven using Pumping Lemma for CFLs; Barbosa, 2018):

$$L = \{\omega \in \{a,b,c\}^* \mid a^n b^n c^n, n \geq 0\}$$

A PDA cannot recognize this language by its definition. However, the QA accepter defined in Figure 4 can (the initial queue symbol is implicitly enqueued after # is dequeued).

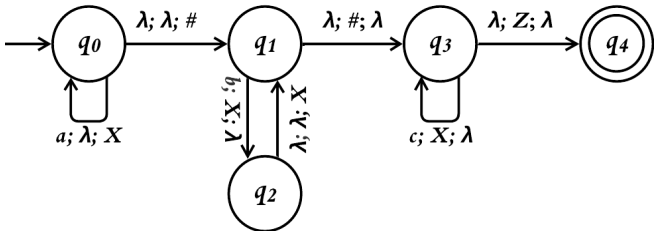


Figure 4. State diagram for the QA that recognizes $L = \{\omega \in \{a,b,c\}^* \mid a^n b^n c^n, n \geq 0\}$

This demonstrates that QAs can recognize languages beyond Context-Free Languages, whereas PDAs cannot. Therefore, QAs are strictly more powerful than PDAs, which means that $L(CFG) \subset L(QA)$.

The power of QA in relation to TM

Finally, since we have established that QAs can access both ends of their queue, it follows that they have computational power beyond that of PDAs. We now prove that QAs are equivalent to Turing Machines (TM), using queue symbol markers that mark each end of the input/output string in tape.

A TM can be formally defined (Hopcroft et al., 2001) by the 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$, where:

- Q is a finite set of states
- Σ is a finite set of input symbols, $\sqcup \notin \Sigma$
- Γ is the tape alphabet, $\Sigma \subseteq \Gamma$
- $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$ is the transition function, which specifies the next state, the symbol to write, and the direction of tape head (left or right)
- $q_0 \in Q$ is the initial state
- $\sqcup \in \Gamma$ is the blank symbol
- $F \subseteq Q$ is a set of accepting states

Given any TM $M_a = (Q_a, \Sigma_a, \Gamma_a, \delta_a, q_{0_a}, \sqcup, F_a)$, we can construct an equivalent QA $M_b = (Q_b, \Sigma_b, \Gamma_b, \delta_b, q_{0_b}, Z_0, F_b)$ as follows:

- $Q_b = Q_a \cup Q_{enq} \cup Q_{deq}$ (additional states used to "shift")
- $\Sigma_b = \Sigma_a$ (input symbols remain unchanged)
- $\Gamma_b = \Gamma_a \cup \{\#, \$\}$ (queue symbols and special markers)
- $q_{0_b} = q_{0_a}$ (initial state remains unchanged)
- $Z_0 = Z$ (initial queue symbol remains unchanged)
- $F_b = F_a$ (final states remain unchanged)
- $\forall q \in Q_a, \forall s \in \Sigma_a \forall \gamma \in \Gamma_a, \forall (q', \gamma, d) \in \delta_a(q, s)$,

- **Tape movement in a Queue:** Since M_a can freely move its head infinitely in both directions of the tape, M_b must adjust its queue head to what the tape head shifts to each time. We do this by having the location of the tape head within the input string correlate to the head of the queue, with special markers # and $ that say the leftmost/rightmost symbol of the tape is to their left in the queue.

  Have M_b queue the entire input string ω, where # is queued right after first symbol of ω and is $ queued after the last symbol of ω of the queue. Also, assume that the head of the queue is the leftmost input symbol upon the start of TM computation.

  We can simulate the TM in QA by:

  1. When the tape moves *left*,
     (a) $Left+1$ and not queue head
        – Dequeue everything
        – Enqueue everything until head of the queue is the $Left+1$ symbol
     (b) $Left+n$ and not queue head
        – Dequeue everything
        – Enqueue everything until head of the queue is the $Left+n$ symbol
     (c) $Left+n$ on head symbol
        – Enqueue n blank spaces
        – Enqueue # after first n blank space
        – Dequeue all symbols
        – Enqueue the same symbols, except former head does not have #
  2. When the tape moves *right*,
     (a) $Right+1$ and not queue tail
        – Dequeue all symbols until $Right+1$ symbol
        – Enqueue everything that was dequeued, making $Right+1$ symbol as head of queue
     (b) $Right+n$ and not queue tail
        – Dequeue all symbols until $Right+n$ symbol
        – Enqueue everything that was dequeued, making $Right+n$ symbol as head of queue
     (c) $Right+n$ on tail symbol
        – Dequeue everything until $
        – Enqueue everything dequeued
        – Enqueue n blank symbols
        – Enqueue $ after n^{th} blank symbol
        – Dequeue $

M_b will only accept it if and only if it reaches a final state in M_a. Therefore, any language recognized by a TM is also recognized by a QA, implying that $L(TM) \subseteq L(QA)$. Since TMs are equivalent to Recursively enumerable (Unrestricted) grammars (Linz, 2012), it follows by transitivity that QAs can recognize Recursively enumerable languages and is Turing-complete.

We now show that construction is possible the other way around. Given any QA $M_b = (Q_b, \Sigma_b, \Gamma_b, \delta_b, q_{0_b}, Z_0, F_b)$, we can construct an equivalent TM $M_a = (Q_a, \Sigma_a, \Gamma_a, \delta_a, q_{0_a}, \sqcup, F_a)$ as follows:

- $Q_a = Q_b \cup Q_{Q_{enq} \cup Q_{deq}}$ (states in M_b and new set of states that facilitate queue-like operations in tape)
- $\Sigma_a = \Sigma_b$
- $\Gamma_a = \Gamma_b \cup \Sigma_b \cup \{\sqcup\}$ (input symbols in M_b and the blank symbol)
- $q_{0_b} = q_{0_a}$
- $F_a = F_b$
- $\forall q \in Q_b, \forall s \in \Sigma_b, \forall \gamma \in \Gamma_b, \forall (q', \gamma') \in \delta_b(q, s, \gamma)$,
$$\delta_a(q,s) = \begin{cases} (q_{\text{enq}}, s, R) & \text{if } \gamma' \neq \lambda \text{ (enqueue)} \\ (q_{\text{deq}}, s, L) & \text{if } \gamma \neq \lambda \text{ (dequeue)} \\ (q_{\text{lmd}}, s, R) & \text{if } \gamma = \lambda \wedge \gamma' = \lambda \text{ (λ transition)} \end{cases}$$
- **Queue Operations in a TM Tape:** Since M_b stores its input in a queue, M_a simulates queue behavior using a single tape:

  1. Enqueue γ:
     – $\delta_a(q_{\text{enq}}, \sqcup) = (q, \gamma, R)$ If the tape head reads a blank symbol ($\sqcup$), write γ and move right
     – $\delta_a(q_{\text{enq}}, s) = (q_{\text{enq}}, s, R), \quad s \neq \sqcup$ Otherwise, move right to find the first blank symbol, then write γ and move right again
  2. Dequeue γ:
     – $\delta_a(q_{\text{deq}}, s) = (q, s, L)$ Move tape head left until $\sqcup$ is encountered
     – $\delta_a(q_{\text{deq}}, \sqcup) = (q_{deq1}, \sqcup, R)$ Once a blank symbol is read, move right
     – $\delta_a(q_{\text{deq1}}, \gamma) = (q, \sqcup, R)$ Overwrite γ (should be the last symbol in input) with $\sqcup$ on the tape head and move right (keep tape head in queue)
  3. λ-transitions:
     – $\delta_a(q_{\text{lmd}}, \gamma) = (q, \gamma, L)$ Given that we move the tape head to the right before this, move back left to preserve the position its in.
  4. Empty queue state:
     – Once initial queue symbol Z is read (assuming Z was explicitly written in), the queue is empty

Transitions in M_a to correspond to δ_b in M_b. Transitions that don't touch the queue will typically be the same, but dequeuing and enqueuing will need extra states to handle the movement of tape from both ends of the queue. Acceptance occurs when M_b reaches a final state and the queue is empty, ensuring that M_b correctly recognizes the same language as M.

Since we have shown both $L(TM) \subseteq L(QA)$ and $L(QA) \subseteq L(TM)$, it follows that $L(TM) = L(QA)$ Thus, QAs are Turing-equivalent.

In summation, the computational power of QA is equivalent to that of a TM (QA is Turing-equivalent), as it can recognize Type-0 languages in the Chomsky hierarchy, formally expressed as $L(QA) = L(TM)$.

APPLICATION

In this section, the paper will discuss an application of the Queue automaton in a professional field. For this case, the paper will dive into the field of Bioinformatics. Imagine a computer that instead of circuitry, it is composed of DNA, proteins and the like that make the human biology. This is what they call in the field as a "biological computer". (Sakowski et al., 2022), in their article "DNA Computing: Concepts for Medical Applications", suggests a queue automata is the best applied computational model for the biological computer.

They state that the queue automata is best and preferred because of its ability to task schedule controlled genome sequence reading on a molecular level. In specific, they use the IIB restriction endonuclease.

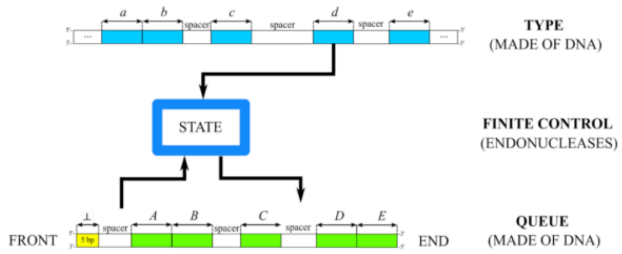


Figure 5. Diagram of Biological Computer

Figure 5. shows the diagram of the proposed queue automaton by (Sakowski et al., 2022). Here, they define that:

- $\Sigma = (a,b,c,d)$
- $\Gamma = (A,B,C,D,E,\perp)$
- $Z_0 = \perp$, where $Z_0 \in \Gamma$

In less mathematical terms, **Figure 5** has the input symbols Σ containing a,b,c,d. Then the queue symbols which will go into the tape Γ which contains $A,B,C,D,E,\perp$ then lastly, the initial queue symbol, $\perp$ which is part of the finite set of queue symbols.

However, this still lacks all the tuples defined in the Machine Definition section. At this point, the paper will try to abridge biology definitions with the definitions set for the computational model.

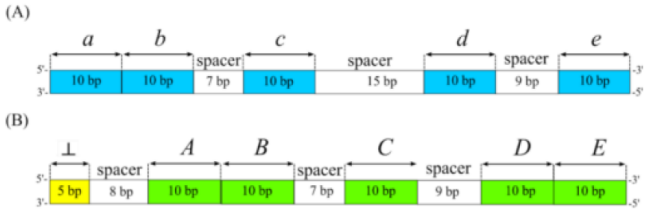


Figure 6. DNA Sequence

The paper will now refer to **Figure 6.** to better see outside of isolation the different tuples. In addition to the earlier defined tuples, here **Figure 6.** can be defined as such:

1. q_0 is the first cut place of the type IIB restriction, i.e., the entire sequence from a to d. This can also be seen in **Figure 5.**
2. $F \in Q$ are the places generated after the last cut.
3. Γ are encoded in the DNA fragment during transition molecule. These are only dequeued after the transition relates to that specific queue symbol.
4. Z is a separate DNA molecule which is a unique fragment from other
5. δ are the transitions encoded in the DNA chain. Queue symbols are placed in the left and at the right side of the DNA structure.
6. The term "SPACER" is a lambda or empty transition.

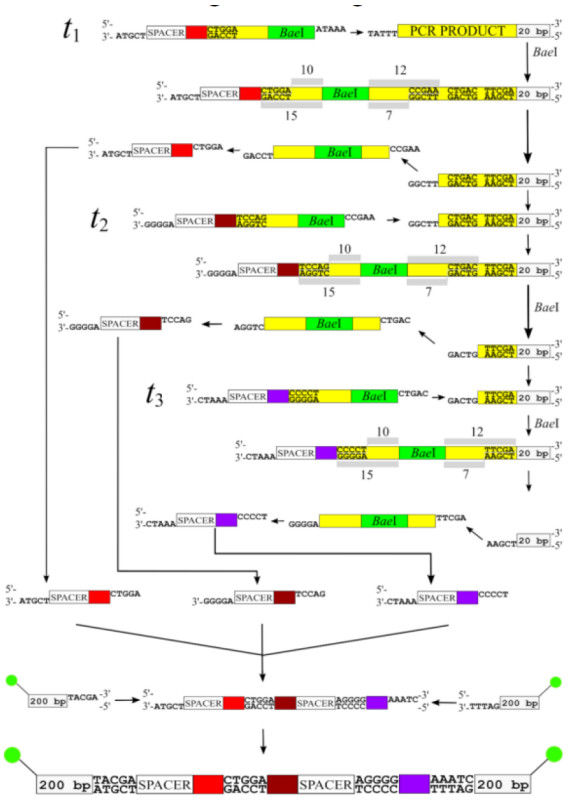


Figure 7. Bael Enzyme

Figure 7. is an entire queue automata within the concept of the "biological computer". Here, the cuts are t_1, t_2, t_3. To formally finish the application, the final or accepting state is, as defined, the last DNA sequence after the final cut.

In **Figure 7.**, the final cut is the very last, very bottom sequence where the queue is now empty. This is defined as the Polymerase Chain Reaction. There is now a copy of the DNA segment, after having used the Bael enzyme act as the queue where information is stored and dequeued after adding each fragment to complete the sequence.

References

Barbosa, H. (2018). Cs:4330 theory of computation context-free languages non-context-free languages [accessed on: 2025-03-25]. https://homepages.dcc.ufmg.br/~hbarbosa/teaching/uiowa/toc/notes/09-non-cfl.pdf

Critchlow, C., & Eck, D. (2011). Foundations of computation. https://eng.libretexts.org/Bookshelves/Computer_Science/Programming_and_Computation_Fundamentals/Foundations_of_Computation_(Critchlow_and_Eck)

EITCA. (2024). What does it mean that one language is more powerful than another? https://eitca.org/cybersecurity/eitc-is-cctf-computational-complexity-theory-fundamentals/context-sensitive-languages/chomsky-hierarchy-and-context-sensitive-languages/what-does-it-mean-that-one-language-is-more-powerful-than-another/

Fitch, W. T. (2014). Toward a computational framework for cognitive biology: Unifying approaches from cognitive neuroscience and comparative cognition. *Physics of Life Reviews*, *11*. https://doi.org/10.1016/j.plrev.2014.04.005

GeeksforGeeks. (2025). Equivalence in theory computation. https://www.geeksforgeeks.org/equivalence-in-theory-of-computation/

Gopalakrishnan, G. (2006). *Nfa and regular expressions. in: Computation engineering*. Springer Science Business Media. https://doi.org/10.1007/0-387-32520-4_9

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2001). *Introduction to automata theory, languages, and computation* (Vol. 32). ACM New York, NY, USA. https://www-2.dc.uba.ar/staff/becher/Hopcroft-Motwani-Ullman-2001.pdf

Jakobi, S., Meckel, K., Mereghetti, C., & Palano, B. (2021). The descriptional power of queue automata of constant length. *Acta Informatica*, *58*. https://doi.org/10.1007/s00236-021-00398-7

Jan, H. (2012). How can one simulate a pda with a fifo queue pda? [version: 2017-04-13]. https://cs.stackexchange.com/q/7004

Linz, P. (2012). *An introduction to formal languages and automata*. Jones & Bartlett Learning. https://fall14cs.wordpress.com/wp-content/uploads/2017/04/an-introduction-to-formal-languages-and-automata-5th-edition-2011.pdf

Minea, M. (2019). Compsci 501: Formal language theory lecture 5: Nonregular languages. pumping lemma. https://people.cs.umass.edu/~marius/class/cs501/lec5-nup.pdf

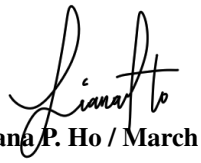
Sakowski, S., Waldmajer, J., Majsterek, I., & Poplawski, T. (2022). Dna computing: Concepts for medical applications. *Applied Sciences*, *12*(14). https://doi.org/10.3390/app12146928

APPENDIX A: RECORD OF CONTRIBUTION

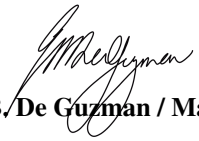
Group Members:

- P1: De Guzman, Evan Mari
- P2: Ho, Denise Liana

Activity	P1	P2
Topic Formulation	30.0	70.0
Machine Definition	40.0	60.0
Formal Language and Computational Power	5.0	95.0
Applications	90.0	10.0
Raw Total	165.0	235.0
TOTAL	**41.25**	**58.75**



Denise Liana P. Ho / March 27, 2025



Evan Mari B. De Guzman / March 27, 2025