Time Complexity

The distinction between the past, present and future is only a stubbornly persistent illusion.

(Albert Einstein)
Consider: $A = \{0^k1^k \mid k \geq 0\}$

What kind of language is this?

Is it **decidable**?

How much **time** does a single tape Turing Machine need in order to decide this language?

Low-level description given:

$M_1 =$ “On input string $w$:
1. Scan across the tape and **reject** if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, reject. Otherwise, if neither 0s nor 1s remain on the tape, accept.”
“Let $M$ be a **deterministic** Turing machine that **halts** on all inputs.

The **running time** or **time complexity** of $M$ is the function $f : \mathbb{N} \rightarrow \mathbb{N}$, where $f(n)$ is the maximum number of steps that $M$ uses on any input of length $n$.

If $f(n)$ is the running time of $M$, we say that $M$ runs in time $f(n)$ and that $M$ is an $f(n)$ **time Turing machine**.

Customarily we use $n$ to represent the length of the **input**.”

**Types of analysis:**

- **Worst-case analysis**: longest running time of all inputs of a particular length.
- **Average-case analysis**: average of all the running times of inputs of a particular length.

We will focus on Worst-case analysis.
Big-O Notation

Technique for estimating an otherwise complex analysis of algorithm performance.

Asymptotic analysis: Focuses on the performance of the algorithm on large inputs.

- Generally only care about the highest-order term, because they dominate.
- Ignore coefficients.

Example: $f(n) = 6n^3 + 2n^2 + 20n + 45$

Big-O (asymptotic) notation: $f(n) = O(n^3)$

DEFINITION:
“Let $f$ and $g$ be functions $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$. Say that $f(n) = O(g(n))$ if positive integers $c$ and $n_0$ exist such that for every integer $n \geq n_0$, $f(n) \leq cg(n)$.

When $f(n) = O(g(n))$, we say that $g(n)$ is an upper bound for $f(n)$, or more precisely, that $g(n)$ is an asymptotic upper bound for $f(n)$, to emphasize that we are suppressing constant factors.”
Big-O Examples

Let $f_1(n) = 5n^3 + 2n^2 + 22n + 6$.
- Highest order term: $5n^3$
- Disregarding the coefficient, now is: $n^3$
- $f_1(n) = O(n^3)$
- If $c = 6$ and $n_0 = 10$. Then: $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ for every $n \geq 10$
- Is $f_1(n) = O(n^4)$ true?
- Is $f_1(n) = O(n^2)$ true?

A log is a log.
- Math identity: $\log_b n = \log_2 n / \log_2 b$
- For $f(n) = O(\log n)$, no need to give the base.

Let $f_2(n) = 3n \log_2 n + 5n \log_2 \log_2 n + 2$.
- $f_2(n) = O(n \log n)$

Let $f_3(n) = O(n^2) + O(n)$.
- $O()$ represents suppressed constants.
- $f_3(n) = O(n^2)$.

Let $f_4(n) = 2^{O(\log n)}$. Is it polynomial or exponential?
- Math identity: $n = 2^{\log_2 n}$, thus $n^c = 2^{c \log_2 n}$.
- $f_4(n) = 2^{O(\log n)} = n^c$ for some constant $c$.
- $n^c$ is a polynomial bound.

What does $n^{O(1)}$ simplify to?
- (i.e., Is it linear, polynomial, or exponential?)
- $n^{O(1)} = n^c$

Conversely, exponential bounds are bounds in the form $2^{(n^\delta)}$ when $\delta$ is a real number greater than 0.
Small-O Notation

**DEFINITION:**
“Let \( f \) and \( g \) be functions \( f, g : \mathbb{N} \to \mathbb{R}^+ \).

Say that \( f(n) = o(g(n)) \) if \( \lim_{n \to \infty} \left[ \frac{f(n)}{g(n)} \right] = 0. \)

In other words, \( f(n) = o(g(n)) \) means that for any real number \( c > 0 \), a number \( n_0 \) exists, where \( f(n) < cg(n) \) for all \( n \geq n_0 \).”

**Intuitive understanding:**
Difference between **big-O** and **small-o** is analogous to the difference between \( \leq \) and \(<\).

**Observe:**
1. \( \sqrt{n} = o(n) \).
2. \( n = o(n \log \log n) \).
3. \( n \log \log n = o(n \log n) \).
4. \( n \log n = o(n^2) \).
5. \( n^2 = o(n^3) \).

However, \( f(n) \) is never \( o(f(n)) \).
Asymptotic behavior dominates as $n$ goes to $\infty$. 

**Time Complexity Visualized**
Algorithm Analysis

Analyze each stage individually:

- **Stage 1**: Takes $2n$ steps (scan & return head to beginning), so is $O(n)$.
- **Stage 2 & 3**: Each scan $[O(n)]$ crosses off a 0 and its corresponding 1.
  - Can only repeat max of $n/2$ times.
  - Therefore, is $O(n^2)$.
- **Stage 4**: Single scan, therefore $O(n)$.

Total time for $M_1$ is $O(n)+O(n^2)+O(n)$, or $O(n^2)$.

Consider: $A = \{0^k1^k \mid k \geq 0\}$.

$M_1$ = “On input string $w$:
1. Scan across the tape and reject if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, reject. Otherwise, if neither 0s nor 1s remain on the tape, accept.”
**Time Complexity Classes**

**DEFINITION:**
“Let \( t : \mathcal{N} \rightarrow \mathcal{R}^+ \) be a function.

Define the \textit{time complexity class}, \( \text{TIME}(t(n)) \), to be the collection of all languages that are \textit{decidable} by an \( O(t(n)) \) time Turing machine.”

**INTUITION:**
This is different from the classes of languages previously discussed, which focused on \textit{computability}.

This classification focuses on the \textit{time resources} required to perform the decision.

**APPLICATION:**
Let \( A = \{ \theta^k 1^k | k \geq 0 \} \).

Analysis shows that \( A \in \text{TIME}(n^2) \) because \( M_1 \) \textit{decides} \( A \) in time \( O(n^2) \) and \( \text{TIME}(n^2) \) contains all languages that can be decided in \( O(n^2) \) time.

\textbf{BUT}

Is there a machine that decides \( A \) asymptotically more quickly?

In other words, is \( A \) in \( \text{TIME}(t(n)) \) for \( t(n) = o(n^2) \)?
Improving Asymptotic Performance of $M_1$

**Question:** Would it help to cross off two 0s and two 1s in each pass?

**Different approach:** Remove half of the 0s and half of the 1s in each pass.

**Question:** How many times can you divide something in half before running out of discrete numbers?

Help for understanding $M_2$:
- Even # + Even # = Even #.
- Odd # + Odd # = Even #.
- Odd # + Even # = Odd #.

$M_2 = \text{“On input string } w\text{ :} $

1. Scan across the tape and reject if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, reject.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, accept. Otherwise, reject.”
Improving Asymptotic Performance of $M_1$ (cont.)

Stage 1, 2, 3, 4, & 5 each take $O(n)$.

How many times does Stage 2 loop?
- Stage 4 crosses off at least half of the 1s and 0s each time through the loop, so repeat is a maximum of $1 + \log_2 n$.
- Total time for Stage 2-4 is $(1 + \log_2 n)O(n)$, or $O(n \log n)$.

Total time complexity:

$O(n) + O(n \log n) = O(n \log n)$.

$\mathcal{A} \in \text{TIME}(n \log n)$

Important question: Can the Asymptotic performance be improved further?

$M_2$ = “On input string $w$:
1. Scan across the tape and reject if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, reject.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, accept. Otherwise, reject.”
Improving Asymptotic Performance of $M_1$ (cont.)

Time complexity?

$O(n)$ (a.k.a., linear time)
Therefore, $A \in \text{TIME}(n)$ must also be true.

**Problem:** It’s not possible to compute $A$ in $O(n)$ on a single tape turing machine (see problem 7.49).

**Important Realization:** Time complexity is dependant on the model of computation. It is an analysis of the machine (algorithm) that performs the computation.

**Church-Turing thesis** implies that all “reasonable” models of computation are equivalent.

**In practice:** We will discuss how important (or not) this will be for our classification system in upcoming sections.

$M_3 =$ “On input string $w$:

1. Scan across tape 1 and **reject** if a 0 is found to the right of a 1.
2. Scan across the 0s on tape 1 until the first 1. At the same time, copy the 0s onto tape 2.
3. Scan across the 1s on tape 1 until the end of the input. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0s are crossed off before all the 1s are read, **reject**.
4. If all the 0s have now been crossed off, **accept**. If any 0s remain, **reject**.”
Complexity Relationships: Single-tape vs. Multi-tape TM

THEOREM:
“Let \( t(n) \) be a function, where \( t(n) \geq n \). Then every \( t(n) \) time multitape Turing machine has an equivalent \( O(t^2(n)) \) time single-tape Turing machine.”

Remember Proof for conversion of multi-tape to single-tape TM.

Let \( M \) be a \( k \)-tape TM that runs in \( t(n) \) time. We construct a single-tape TM \( S \) that runs in \( O(t^2(n)) \) time.

(From Theorem 3.13) For each simulation step:
- \( S \) must scan the tape twice.
- When moving any head right, if it would move off the allocated space, then the contents of \( S \)'s tape must be shifted right before continuing.

Most important question: What is the maximum length of all of \( S \)'s tape? Why?

Answer: \( O(t(n)) \)
Known values:

- \(M\) runs in \(O(t(n))\).
- \(S\)’s tape is a max length of \(O(t(n))\).
- For each step of \(S\)’s simulation,
  - \(S\) scans it’s tape to determine the current state, which is \(O(t(n))\).
  - \(S\) scans it’s tape to apply the transitions:
    - If moving to the right, and everything must be shifted, complexity for this single operation is \(O(t(n))\).
    - Shift happens at most \(k\) times.
    - \(k\) is a constant!
  - Each step takes \(O(t(n))\) time to simulate.
    - Is this obvious to you?
- Total number of steps that \(S\) must simulate: \(O(t(n))\).

Putting it all together:

- \(S\) initializing its tape: \(O(n)\).
- \(S\) performing the simulation: \(O(t^2(n))\).
- Total time: \(O(n) + O(t^2(n)) = O(t^2(n))\).

We have assumed that \(t(n) \geq n\).

Is this a valid assumption?

Proof is complete.
Let $N$ be a nondeterministic Turing machine that is a decider. The running time of $N$ is the function $f : \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the maximum number of steps that $N$ uses on any branch of its computation on any input of length $n$, as shown in the figure.

**Note:** Even though we are defining a running time for nondeterministic TMs, this does not imply that it corresponds to any real-world device. It is a theoretical tool which we use to understand computational problems.
Complexity Relationships: Multi-tape TM vs. Non-deterministic TM (cont)

**THEOREM:**
“Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.”

**PROOF:**
Let $N$ be a nondeterministic TM running in $t(n)$ time. Construct a deterministic TM $D$ that simulates $N$ as in the proof of Theorem 3.16 by searching $N$’s nondeterministic computation tree. Analysis follows.

**Known Facts:**
- On an input of length $n$, every branch of $N$’s nondeterministic computation tree has a length of at most $t(n)$.
- Every node has at most $b$ children.
- Total # of leaves is at most $b^{t(n)}$.
- Total # of nodes is less than twice the # of leaves, so bound by $O(b^{t(n)})$.
- Time to start at root and travel down is $O(t(n))$.

Running Time for $D$ is $O(t(n)b^{t(n)}) = 2^{O(t(n))}$.

$D$ is a multi-tape TM (according to Theorem 3.16). What is the complexity when converted to single-tape?
Does $O(t(n)b^{t(n)}) = 2^{O(t(n))}$?

Let $b > 1$ be a constant. Show that $O(t(n)) \times O(b^{t(n)}) = 2^{O(t(n))}$.

**Answer:** Let $f_1(n) = O(t(n))$ and $f_2(n) = O(b^{t(n)})$, so we want to show that $f_1(n)f_2(n) = 2^{O(t(n))}$. Because $f_1(n) = O(t(n))$, there exist constants $c_1$ and $n_1$ such that $f_1(n) \leq c_1 t(n)$ for all $n \geq n_1$. Because $f_2(n) = O(b^{t(n)})$, there exist constants $c_2$ and $n_2$ such that $f_2(n) \leq c_2 b^{t(n)}$ for all $n \geq n_2$. Consequently, letting $c_0 = c_1 c_2$, we get

$$f_1(n) f_2(n) \leq c_0 t(n) b^{t(n)}$$

for all $n \geq n_0 \equiv \max(n_1, n_2)$. Now recall that $x = 2^{\log_2(x)}$ for any $x$, so

$$c_0 t(n) b^{t(n)} = 2^{\log_2(c_0 t(n) b^{t(n)})} = 2^{\log_2(c_0) + \log_2(t(n)) + t(n) \log_2(b)} = 2^{O(t(n))}$$

because $\log_2(t(n)) = O(t(n))$. 