Teoria degli Algoritmi

Corso di Laurea Magistrale in Matematica Applicata
a.a. 2020-21

Gabriele Tolomei

Dipartimento di Informatica
Sapienza Università di Roma
tolomei@di.uniroma1.it

Lecture 5: Complexity

Table of Contents

Table of Contents

Computational Complexity

- We have seen that there exist some problems that are **not decidable**, **nor even recognizable**

Computational Complexity

- We have seen that there exist some problems that are **not decidable**, **nor even recognizable**
- Even though a problem is decidable, and thus computationally solvable in theory, it may not be solvable **in practice**

Computational Complexity

- We have seen that there exist some problems that are **not decidable**, **nor even recognizable**
- Even though a problem is decidable, and thus computationally solvable in theory, it may not be solvable **in practice**
- The reason of such **intractability** is related to the extraordinary amount of resources (mostly, time and space/memory) required by a solution to the problem

Computational Complexity

- We have seen that there exist some problems that are **not decidable**, **nor even recognizable**
- Even though a problem is decidable, and thus computationally solvable in theory, it may not be solvable **in practice**
- The reason of such **intractability** is related to the extraordinary amount of resources (mostly, time and space/memory) required by a solution to the problem
- We now delve into the realm of **computational complexity theory**, which investigates the time, space/memory, and any other resource needed to solve a computational problem

Complexity

- We start from the most important resource (**time**), presenting the basics of complexity theory

Complexity

- We start from the most important resource (**time**), presenting the basics of complexity theory
- First, we introduce a way of **measuring** the time used to solve a problem

Complexity

- We start from the most important resource (**time**), presenting the basics of complexity theory
- First, we introduce a way of **measuring** the time used to solve a problem
- Then, we show how to classify problems according to the amount of time they required to be solved

Complexity

- We start from the most important resource (**time**), presenting the basics of complexity theory
- First, we introduce a way of **measuring** the time used to solve a problem
- Then, we show how to classify problems according to the amount of time they required to be solved
- After that, we discuss the possibility that certain decidable problems need a huge amount of time therefore making them intractable in practice

Complexity

- We start from the most important resource (**time**), presenting the basics of complexity theory
- First, we introduce a way of **measuring** the time used to solve a problem
- Then, we show how to classify problems according to the amount of time they required to be solved
- After that, we discuss the possibility that certain decidable problems need a huge amount of time therefore making them intractable in practice
- Finally, we try to determine when we are facing with such intractable problems

Table of Contents

Measuring Complexity: An Example

- Consider the language $A = \{0^k 1^k \mid k \geq 0\}$

Measuring Complexity: An Example

- Consider the language $A = \{0^k 1^k \mid k \geq 0\}$
- Obviously, A is a **decidable language**, i.e., we can build a TM that decides A

Measuring Complexity: An Example

- Consider the language $A = \{0^k 1^k \mid k \geq 0\}$
- Obviously, A is a **decidable language**, i.e., we can build a TM that decides A
- How much time does a single-tape TM need to decide A?

Measuring Complexity: An Example

- Consider the language $A = \{0^k 1^k \mid k \geq 0\}$
- Obviously, A is a **decidable language**, i.e., we can build a TM that decides A
- How much time does a single-tape TM need to decide A?
- To answer the question above, let us describe a TM M_A that decides A, so that we can count the **number of steps** it takes

Measuring Complexity: An Example

Example (A decider for $A = \{0^k 1^k \mid k \geq 0\}$)

M_A = "On input string x:

1. Scan across the tape and if a 0 occurs to the right of a 1, **reject**;

Measuring Complexity: An Example

Example (A decider for $A = \{0^k 1^k \mid k \geq 0\}$)

M_A = "On input string x:

1. Scan across the tape and if a 0 occurs to the right of a 1, **reject**;
2. If both 0s and 1s are left on the tape, repeat:

Measuring Complexity: An Example

Example (A decider for $A = \{0^k1^k \mid k \geq 0\}$)

M_A = "On input string x:

1. Scan across the tape and if a 0 occurs to the right of a 1, **reject**;
2. If both 0s and 1s are left on the tape, repeat:
   a. Scan across the tape, crossing off a single 0 and a single 1;

Measuring Complexity: An Example

Example (A decider for $A = \{0^k1^k \mid k \geq 0\}$)

M_A = "On input string x:

1. Scan across the tape and if a 0 occurs to the right of a 1, **reject**;
2. If both 0s and 1s are left on the tape, repeat:
   a. Scan across the tape, crossing off a single 0 and a single 1;
3. If 0s (resp. 1s) still remain after all the 1s (resp. 0s) have been deleted, **reject**; otherwise, if neither 0s nor 1s are left on the tape, **accept**"

Measuring Complexity: An Example

Let's see how M_A works on a specific input $x = 0011$

- **Step 1** Scan all the four cells of the tape: the input is well-formed so we can continue to the next step;

Measuring Complexity: An Example

Let's see how M_A works on a specific input $x = 0011$

- **Step 1** Scan all the four cells of the tape: the input is well-formed so we can continue to the next step;
- **Loop test** Scan all four cells to check if some 0s and 1s are still on the tape: YES!

Measuring Complexity: An Example

Let's see how M_A works on a specific input $x = 0011$

- **Step 1** Scan all the four cells of the tape: the input is well-formed so we can continue to the next step;
- **Loop test** Scan all four cells to check if some 0s and 1s are still on the tape: YES!
- **1st loop iteration** Scan all four cells crossing off a 1 and a 0, leaving on the tape the following string: $X0X1$

Measuring Complexity: An Example

Let's see how M_A works on a specific input $x = 0011$

- **Step 1** Scan all the four cells of the tape: the input is well-formed so we can continue to the next step;
- **Loop test** Scan all four cells to check if some 0s and 1s are still on the tape: YES!
- **1st loop iteration** Scan all four cells crossing off a 1 and a 0, leaving on the tape the following string: $X0X1$
- **Loop test** Scan all four cells to check if some 0s and 1s are still on the tape: YES!

Measuring Complexity: An Example

Let's see how M_A works on a specific input $x = 0011$

- **Step 1** Scan all the four cells of the tape: the input is well-formed so we can continue to the next step;
- **Loop test** Scan all four cells to check if some 0s and 1s are still on the tape: YES!
- **1st loop iteration** Scan all four cells crossing off a 1 and a 0, leaving on the tape the following string: $X0X1$
- **Loop test** Scan all four cells to check if some 0s and 1s are still on the tape: YES!
- **2nd loop iteration** Scan all four cells crossing off a 1 and a 0, leaving on the tape the following string: $XXXX$

Measuring Complexity: An Example

Let's see how M_A works on a specific input $x = 0011$

- **Step 1** Scan all the four cells of the tape: the input is well-formed so we can continue to the next step;
- **Loop test** Scan all four cells to check if some 0s and 1s are still on the tape: YES!
- **1st loop iteration** Scan all four cells crossing off a 1 and a 0, leaving on the tape the following string: $X0X1$
- **Loop test** Scan all four cells to check if some 0s and 1s are still on the tape: YES!
- **2nd loop iteration** Scan all four cells crossing off a 1 and a 0, leaving on the tape the following string: $XXXX$
- **Loop test** Scan all four cells to check if some 0s and 1s are still on the tape: NO! $\rightarrow$ **accept!**

Measuring Complexity: Factors

- The number of steps that an algorithm performs on a particular input depends on many factors

Measuring Complexity: Factors

- The number of steps that an algorithm performs on a particular input depends on many factors
- For example, if the input is a graph, the number of steps may depend on the number of nodes, the number of edges, or some combination of those

Measuring Complexity: Factors

- The number of steps that an algorithm performs on a particular input depends on many factors
- For example, if the input is a graph, the number of steps may depend on the number of nodes, the number of edges, or some combination of those
- For the sake of simplicity, we compute the **running time** of an algorithm purely as a function of the string representing the input

Measuring Complexity: Factors

- The number of steps that an algorithm performs on a particular input depends on many factors
- For example, if the input is a graph, the number of steps may depend on the number of nodes, the number of edges, or some combination of those
- For the sake of simplicity, we compute the **running time** of an algorithm purely as a function of the string representing the input
- In **worst-case analysis**, which we focus on here, we consider the longest running time of **all** inputs of a fixed size

Measuring Complexity: Factors

- The number of steps that an algorithm performs on a particular input depends on many factors
- For example, if the input is a graph, the number of steps may depend on the number of nodes, the number of edges, or some combination of those
- For the sake of simplicity, we compute the **running time** of an algorithm purely as a function of the string representing the input
- In **worst-case analysis**, which we focus on here, we consider the longest running time of **all** inputs of a fixed size
- In **average-case analysis**, instead, we consider the average of all the running times of **all** inputs of a fixed size

The Definition of Running Time

Definition (Running Time (Time Complexity))

Let M be a deterministic Turing machine that halts on all inputs (i.e., a decider).
The **running time** (or **time complexity**) of M is the function:

$$f : \mathbb{N} \mapsto \mathbb{N}$$

where $f(n)$ is the **maximum** number of steps that M needs to perform in order to halt on **any** input of size n.

Table of Contents

Big-O

- The exact running time of an algorithm is often the result of a complex expression

Big-O

- The exact running time of an algorithm is often the result of a complex expression
- We do not need to be extremely accurate, and an estimation of the running time is sufficient

Big-O

- The exact running time of an algorithm is often the result of a complex expression
- We do not need to be extremely accurate, and an estimation of the running time is sufficient
- A convenient form of estimation is called **asymptotic analysis**

Big-O

- The exact running time of an algorithm is often the result of a complex expression
- We do not need to be extremely accurate, and an estimation of the running time is sufficient
- A convenient form of estimation is called **asymptotic analysis**
- Such an estimation tries to capture the running time of an algorithm when this is input with **large size** inputs

Big-O

- We consider **only** the highest order terms of the expression for the running time of the algorithm

Big-O

- We consider **only** the highest order terms of the expression for the running time of the algorithm
- Therefore, we discard both the coefficient of that largest term as well as any lower order terms

Big-O

- We consider **only** the highest order terms of the expression for the running time of the algorithm
- Therefore, we discard both the coefficient of that largest term as well as any lower order terms
- The rationale of this choice is that, intuitively, for very large inputs the highest order term dominates over the others

Big-O

- We consider **only** the highest order terms of the expression for the running time of the algorithm
- Therefore, we discard both the coefficient of that largest term as well as any lower order terms
- The rationale of this choice is that, intuitively, for very large inputs the highest order term dominates over the others

Example

Let $f(n) = 6n^3 + 2n^2 + 20n + 45$. This is a 4-term polynomial expression, and the highest order term is $6n^3$. Disregarding the coefficient 6, we say that $f(n)$ is **asymptotically** at most n^3

Big-O: Definition

- We formalize the **asymptotic notation** (or **Big-O notation** as follows:

Big-O: Definition

- We formalize the **asymptotic notation** (or **Big-O notation** as follows:

Definition (Big-O)

Let f, g be two functions such that $f, g : \mathbb{N} \mapsto \mathbb{R}_{\geq 0}$. We say that $\mathbf{f(n) = O(g(n))}$ if there exist $c, n_0 \in \mathbb{Z}^+$, such that for every $n \geq n_0$:

$$f(n) \leq cg(n)$$

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 stress the fact that we are not considering any constant factor.

Big-O: Intuition

- Intuitively, $f(n) = O(g(n))$ means that f is less than or equal to g if we disregard differences up to a constant factor

Big-O: Intuition

- Intuitively, $f(n) = O(g(n))$ means that f is less than or equal to g if we disregard differences up to a constant factor
- We can think of O as representing a suppressed constant

Big-O: Intuition

- Intuitively, $f(n) = O(g(n))$ means that f is less than or equal to g if we disregard differences up to a constant factor
- We can think of O as representing a suppressed constant
- In practice, most functions f that we will encounter have an obvious highest order term

Big-O: Examples

Example

Let $f(n) = 5n^3 + 2n^2 + 22n + 6$.

Big-O: Examples

Example

Let $f(n) = 5n^3 + 2n^2 + 22n + 6$.
Thus, selecting the highest order term $(5n^3)$ and disregarding its coefficient (5) gives us $f(n) = O(n^3)$.

Big-O: Examples

Example

Let $f(n) = 5n^3 + 2n^2 + 22n + 6$.
Thus, selecting the highest order term ($5n^3$) and disregarding its coefficient (5) gives us $f(n) = O(n^3)$.
Let's verify that this result satisfies our formal definition of asymptotic upper bound. We must find c and n_0, such that:

$$\underbrace{5n^3 + 2n^2 + 22n + 6}_{f(n)} \leq cn^3 \ \forall n \geq n_0$$

Big-O: Examples

Example

Let $f(n) = 5n^3 + 2n^2 + 22n + 6$.
Thus, selecting the highest order term ($5n^3$) and disregarding its coefficient (5) gives us $f(n) = O(n^3)$.
Let's verify that this result satisfies our formal definition of asymptotic upper bound. We must find c and n_0, such that:

$$\underbrace{5n^3 + 2n^2 + 22n + 6}_{f(n)} \leq cn^3 \ \forall n \geq n_0$$

For $c = 6$ and $n_0 = 10$, it holds that:

$$5n^3 + 2n^2 + 22n + 6 \leq 6n^3 \ \forall n \geq 10$$

Big-O: Observations

Note

In the example above, it also holds (trivially) that $f(n) = O(n^4)$, because $n^4 \geq n^3$ and so it is still an asymptotic upper bound of f.
Conversely, $f(n) \neq O(n^2)$: regardless of the values we assign to c and n_0, the condition we seek for according to the definition remains unsatisfied

Big-O: Observations

- Big-O interacts with logarithms in a particular way

Big-O: Observations

- Big-O interacts with logarithms in a particular way
- Usually, when we use logarithms we must specify the **base**, e.g., $x = \log_2 n$

Big-O: Observations

- Big-O interacts with logarithms in a particular way
- Usually, when we use logarithms we must specify the **base**, e.g., $x = \log_2 n$
- The base-2 in the equation above indicates that it is equivalent to $2^x = n$

Big-O: Observations

- Big-O interacts with logarithms in a particular way
- Usually, when we use logarithms we must specify the **base**, e.g., $x = \log_2 n$
- The base-2 in the equation above indicates that it is equivalent to $2^x = n$
- Changing the value of the base b changes the value of $\log_b n$ by a **constant factor**, as it holds that

$$\log_b n = \frac{\log_2 n}{\log_2 b}$$

Big-O: Observations

- Big-O interacts with logarithms in a particular way
- Usually, when we use logarithms we must specify the **base**, e.g., $x = \log_2 n$
- The base-2 in the equation above indicates that it is equivalent to $2^x = n$
- Changing the value of the base b changes the value of $\log_b n$ by a **constant factor**, as it holds that

$$\log_b n = \frac{\log_2 n}{\log_2 b}$$

- When we write $f(n) = O(\log n)$ we don't need to specify the base as constant factors are suppressed anyway

Big-O: Observations

- Big-O notation also appears in arithmetic expressions, e.g., $f(n) = O(n^2) + O(n)$

Big-O: Observations

- Big-O notation also appears in arithmetic expressions, e.g., $f(n) = O(n^2) + O(n)$
- In that case, each occurrence of the O symbol represents a different suppressed constant factor

Big-O: Observations

- Big-O notation also appears in arithmetic expressions, e.g., $f(n) = O(n^2) + O(n)$
- In that case, each occurrence of the O symbol represents a different suppressed constant factor
- In the example, since $O(n^2)$ dominates $O(n)$, we can simply write $f(n) = O(n^2)$

Big-O: Observations

- Big-O notation also appears in arithmetic expressions, e.g., $f(n) = O(n^2) + O(n)$
- In that case, each occurrence of the O symbol represents a different suppressed constant factor
- In the example, since $O(n^2)$ dominates $O(n)$, we can simply write $f(n) = O(n^2)$
- Similarly, when O occurs at the exponent like $f(n) = 2^{O(n)}$ the same idea applies

Big-O: Observations

- Big-O notation also appears in arithmetic expressions, e.g., $f(n) = O(n^2) + O(n)$
- In that case, each occurrence of the O symbol represents a different suppressed constant factor
- In the example, since $O(n^2)$ dominates $O(n)$, we can simply write $f(n) = O(n^2)$
- Similarly, when O occurs at the exponent like $f(n) = 2^{O(n)}$ the same idea applies
- Here, the expression represents an upper bound of 2^{cn} for some constant c

Big-O: Observations

- Following the same idea, in some analysis it may occur that $f(n) = 2^{O(\log n)}$

Big-O: Observations

- Following the same idea, in some analysis it may occur that $f(n) = 2^{O(\log n)}$
- Notice that $n = 2^{\log_2 n}$, and therefore $n^c = 2^{c \log_2 n}$

Big-O: Observations

- Following the same idea, in some analysis it may occur that $f(n) = 2^{O(\log n)}$
- Notice that $n = 2^{\log_2 n}$, and therefore $n^c = 2^{c \log_2 n}$
- As such, $2^{O(\log n)}$ represents an upper bound for n^c for some constant c

Big-O: Observations

- Following the same idea, in some analysis it may occur that $f(n) = 2^{O(\log n)}$
- Notice that $n = 2^{\log_2 n}$, and therefore $n^c = 2^{c \log_2 n}$
- As such, $2^{O(\log n)}$ represents an upper bound for n^c for some constant c
- Finally, the expression $f(n) = O(1)$ is a "convention" to represent a value that is never more than a constant (i.e., it does not depend on n)

Big-O vs. small-o

- Big-O notation has a companion called **small**-o

Big-O vs. small-o

- Big-O notation has a companion called **small**-o
- Intuitively, Big-O notation says that one function is **asymptotically no greater than** another

Big-O vs. small-o

- Big-O notation has a companion called **small**-o
- Intuitively, Big-O notation says that one function is **asymptotically no greater than** another
- Instead, we use small-o to indicate that a function is **asymptotically less than** another

Big-O vs. small-o

- Big-O notation has a companion called **small-o**
- Intuitively, Big-O notation says that one function is **asymptotically no greater than** another
- Instead, we use small-o to indicate that a function is **asymptotically less than** another
- The difference between Big-O and small-o is the same as the difference between $\leq$ and $<$

small-o: Definition

Definition (small-o)

Let f, g be two functions such that $f, g : \mathbb{N} \mapsto \mathbb{R}_{\geq 0}$. We say that $\mathbf{f(n) = o(g(n))}$ if:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

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

Big-O vs. small-o: Example

Example

Let's go back to $f(n) = 5n^3 + 2n^2 + 22n + 6$.

Big-O vs. small-o: Example

Example

Let's go back to $f(n) = 5n^3 + 2n^2 + 22n + 6$.
We already know that $f(n) = O(n^3)$, i.e., we have found there **exist** $c = 6$ and $n_0 = 10$, such that:

$$f(n) = 5n^3 + 2n^2 + 22n + 6 \leq cn^3 \ \forall n \geq n_0$$

Big-O vs. small-o: Example

Example

Let's go back to $f(n) = 5n^3 + 2n^2 + 22n + 6$.
We already know that $f(n) = O(n^3)$, i.e., we have found there **exist** $c = 6$ and $n_0 = 10$, such that:

$$f(n) = 5n^3 + 2n^2 + 22n + 6 \leq cn^3 \ \forall n \geq n_0$$

However, $f(n) \neq o(n^3)$ as we can find a value c for which **there is no** n_0 such that $f(n) < cg(n)$ for all $n \geq n_0$, e.g., $c = 5$:

$$5n^3 + 2n^2 + 22n + 6 \not< 5n^3 \ \forall n \geq n_0$$

Table of Contents

Analyzing Algorithms: An Example

Let's analyze the algorithm implemented by the TM that decides the language $A = \{0^k 1^k \mid k \geq 0\}$

Example (A decider for $A = \{0^k 1^k \mid k \geq 0\}$)

M_A = "On input string x:

1. Scan across the tape and if a 0 occurs to the right of a 1, **reject**;
2. If both 0s and 1s are left on the tape, repeat:
    a. Scan across the tape, crossing off a single 0 and a single 1;
3. If 0s (resp. 1s) still remain after all the 1s (resp. 0s) have been deleted, **reject**; otherwise, if neither 0s nor 1s are left on the tape, **accept**"

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- In stage 1, the machine scans across the tape to verify that the input is of the form 0^*1^*

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- In stage 1, the machine scans across the tape to verify that the input is of the form 0^*1^*
- Assuming n is the input length, this scan takes n steps

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- In stage 1, the machine scans across the tape to verify that the input is of the form 0^*1^*
- Assuming n is the input length, this scan takes n steps
- Resetting the head to the left-most position on the tape requires additional n steps, thereby $2n$ steps in total

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- In stage 1, the machine scans across the tape to verify that the input is of the form 0^*1^*
- Assuming n is the input length, this scan takes n steps
- Resetting the head to the left-most position on the tape requires additional n steps, thereby $2n$ steps in total
- Using Big-O notation, that means $O(n)$ steps

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- In stages 2 the machine scans across the tape to verify if any 0s or 1s are left on the tape

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- In stages 2 the machine scans across the tape to verify if any 0s or 1s are left on the tape
- This is the test M_A needs to take to check whether it should enter the loop or not

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- In stages 2 the machine scans across the tape to verify if any 0s or 1s are left on the tape
- This is the test M_A needs to take to check whether it should enter the loop or not
- Since this involves a full scan of the tape, it costs n steps

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- Within the loop, M_A scans again across the whole tape

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- Within the loop, M_A scans again across the whole tape
- Therefore, each iteration needs n steps to be performed

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- Within the loop, M_A scans again across the whole tape
- Therefore, each iteration needs n steps to be performed
- The point is: how many loop iterations do we need to do **at most**?

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- Within the loop, M_A scans again across the whole tape
- Therefore, each iteration needs n steps to be performed
- The point is: how many loop iterations do we need to do **at most**?
- Since each scan within every loop iteration crosses off two symbols, at most there will be $n/2$ iterations

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- Within the loop, M_A scans again across the whole tape
- Therefore, each iteration needs n steps to be performed
- The point is: how many loop iterations do we need to do **at most**?
- Since each scan within every loop iteration crosses off two symbols, at most there will be $n/2$ iterations
- Each iteration is composed by two full scans (i.e., one for testing the loop condition and the other for crossing off symbols): $2n = O(n)$

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- Within the loop, M_A scans again across the whole tape
- Therefore, each iteration needs n steps to be performed
- The point is: how many loop iterations do we need to do **at most**?
- Since each scan within every loop iteration crosses off two symbols, at most there will be $n/2$ iterations
- Each iteration is composed by two full scans (i.e., one for testing the loop condition and the other for crossing off symbols): $2n = O(n)$
- Overall, $n/2 * O(n) = O(n^2)$ steps

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- In stage 4, the machine makes a full scan to decide whether it should accept or reject the input

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- In stage 4, the machine makes a full scan to decide whether it should accept or reject the input
- The time taken by this stage is again $O(n)$

Analyzing Algorithms: An Example

To analyze M_A, we consider each stage separately

- In stage 4, the machine makes a full scan to decide whether it should accept or reject the input
- The time taken by this stage is again $O(n)$
- Thus, the total time of execution of M_A on an input x whose length is n is:

$$O(n) + O(n^2) + O(n) = O(n^2)$$

Time Complexity Class: A Formal Definition

Definition (The class TIME(t(n)))

Let $t : \mathbb{N} \mapsto \mathbb{R}_{\geq 0}$ be a function.
We define the **time complexity class TIME(t(n))** as the collection of all languages that are decidable by an $O(t(n))$ time Turing machine

Analyzing Algorithms: An Example

On top of the definition of $TIME(t(n))$, a natural question arises:
Is there a TM that decides the language A asymptotically more quickly?

Analyzing Algorithms: An Example

On top of the definition of $TIME(t(n))$, a natural question arises:
Is there a TM that decides the language A asymptotically more quickly?
In other words, is $A \in TIME(t(n))$, where $t(n) = o(n^2)$?

Analyzing Algorithms: An Example

On top of the definition of $TIME(t(n))$, a natural question arises:
Is there a TM that decides the language A asymptotically more quickly?
In other words, is $A \in TIME(t(n))$, where $t(n) = o(n^2)$?

- One quick improvement to M_A would be to cross off four symbols (i.e., two 0s and two 1s) on every scan, instead of just two

Analyzing Algorithms: An Example

On top of the definition of $TIME(t(n))$, a natural question arises:
Is there a TM that decides the language A asymptotically more quickly?
In other words, is $A \in TIME(t(n))$, where $t(n) = o(n^2)$?

- One quick improvement to M_A would be to cross off four symbols (i.e., two 0s and two 1s) on every scan, instead of just two
- That would improve the running time by a factor of 2 since we would need to do $n/4$ iterations rather than $n/2$

Analyzing Algorithms: An Example

On top of the definition of $TIME(t(n))$, a natural question arises:
Is there a TM that decides the language A asymptotically more quickly?
In other words, is $A \in TIME(t(n))$, where $t(n) = o(n^2)$?

- One quick improvement to M_A would be to cross off four symbols (i.e., two 0s and two 1s) on every scan, instead of just two
- That would improve the running time by a factor of 2 since we would need to do $n/4$ iterations rather than $n/2$
- Unfortunately, this won't affect the asymptotic running time (Remember: constant factors do not count!)

Analyzing Algorithms: An Example

We can design another TM M'_A that still decides A, yet it shows that $A \in TIME(n \log n)$

Example (Another decider for $A = \{0^k 1^k \mid k \geq 0\}$)

M'_A = "On input string x:

1. Scan across the tape and if a 0 occurs to the right of a 1, **reject**;
2. Repeat as long as some 0s and 1s are left on the tape:
    - a Scan across the tape, checking if the total number of 0s and 1s remaining is even or odd: if it is odd **reject**;
    - b Scan again across the tape, crossing off every 0 (resp. 1) in alternating way, starting from the first 0 (resp. 1) encountered;
3. If no 0s (resp. 1s) are left on the tape, **accept**; otherwise, **reject**."

Analyzing Algorithms: An Example

- Before analyzing M'_A, let's verify that it actually decides A

Analyzing Algorithms: An Example

- Before analyzing M'_A, let's verify that it actually decides A
- On stage 2.a M'_A checks on the agreement of the parity of the 0s with the parity of the 1s (if the parities agree, the numbers of 0s and 1s are equal)

Analyzing Algorithms: An Example

- Before analyzing M'_A, let's verify that it actually decides A
- On stage 2.a M'_A checks on the agreement of the parity of the 0s with the parity of the 1s (if the parities agree, the numbers of 0s and 1s are equal)
- On every scan performed in stage 2.b, the number of 0s (resp., 1s) is cut in half and any reminder discarded

Analyzing Algorithms: An Example

- To analyze the running time of M'_A, we first observe that each stage takes $O(n)$ steps (i.e., a full linear scan of the tape)

Analyzing Algorithms: An Example

- To analyze the running time of M'_A, we first observe that each stage takes $O(n)$ steps (i.e., a full linear scan of the tape)
- We must then determine how many times each stage is executed

Analyzing Algorithms: An Example

- To analyze the running time of M'_A, we first observe that each stage takes $O(n)$ steps (i.e., a full linear scan of the tape)
- We must then determine how many times each stage is executed
- Stages 1 and 3 are executed only once, thereby taking $O(n) + O(n) = O(n)$ total time

Analyzing Algorithms: An Example

- To analyze the running time of M'_A, we first observe that each stage takes $O(n)$ steps (i.e., a full linear scan of the tape)
- We must then determine how many times each stage is executed
- Stages 1 and 3 are executed only once, thereby taking $O(n) + O(n) = O(n)$ total time
- Stage 2.b crosses off at least half 0s and half 1s at each iteration, so at most $1 + \log_2 n$ iterations are needed before all symbols get crossed off

Analyzing Algorithms: An Example

- To analyze the running time of M'_A, we first observe that each stage takes $O(n)$ steps (i.e., a full linear scan of the tape)
- We must then determine how many times each stage is executed
- Stages 1 and 3 are executed only once, thereby taking $O(n) + O(n) = O(n)$ total time
- Stage 2.b crosses off at least half 0s and half 1s at each iteration, so at most $1 + \log_2 n$ iterations are needed before all symbols get crossed off
- The total time of stages 2, 2.a, and 2.b is: $(1 + \log_2 n) * O(n) = O(n \log n)$

Analyzing Algorithms: An Example

- To analyze the running time of M'_A, we first observe that each stage takes $O(n)$ steps (i.e., a full linear scan of the tape)
- We must then determine how many times each stage is executed
- Stages 1 and 3 are executed only once, thereby taking $O(n) + O(n) = O(n)$ total time
- Stage 2.b crosses off at least half 0s and half 1s at each iteration, so at most $1 + \log_2 n$ iterations are needed before all symbols get crossed off
- The total time of stages 2, 2.a, and 2.b is: $(1 + \log_2 n) * O(n) = O(n \log n)$
- Putting everything together, the total running time of M'_A is: $O(n) + O(n \log n) = O(n \log n)$

Analyzing Algorithms: An Example

- Earlier on, we have shown that $A \in TIME(n^2)$ but now we have a better upper bound, i.e., $A \in TIME(n \log n)$

Analyzing Algorithms: An Example

- Earlier on, we have shown that $A \in TIME(n^2)$ but now we have a better upper bound, i.e., $A \in TIME(n \log n)$
- This result **cannot** be improved on a single-tape TM

Analyzing Algorithms: An Example

- Earlier on, we have shown that $A \in TIME(n^2)$ but now we have a better upper bound, i.e., $A \in TIME(n \log n)$
- This result **cannot** be improved on a single-tape TM
- However, we can decide the language A in $O(n)$ time (a.k.a. **linear time**) if the TM has a second tape

Analyzing Algorithms: An Example

The following two-tape TM M''_A decides A in $O(n)$ time

Example (Another decider for $A = \{0^k 1^k \mid k \geq 0\}$)

M''_A = "On input string x:

1. Scan across the tape and if a 0 occurs to the right of a 1, **reject**.

Analyzing Algorithms: An Example

The following two-tape TM M''_A decides A in $O(n)$ time

Example (Another decider for $A = \{0^k1^k \mid k \geq 0\}$)

$M''_A =$ "On input string x:

1. Scan across the tape and if a 0 occurs to the right of a 1, **reject**.
2. Scan across the 0s on tape 1 until the first 1; simultaneously, copy the 0s on tape 2.

Analyzing Algorithms: An Example

The following two-tape TM M''_A decides A in $O(n)$ time

Example (Another decider for $A = \{0^k1^k \mid k \geq 0\}$)

M''_A = "On input string x:

1. Scan across the tape and if a 0 occurs to the right of a 1, **reject**.
2. Scan across the 0s on tape 1 until the first 1; simultaneously, copy the 0s on 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**.

Analyzing Algorithms: An Example

The following two-tape TM M''_A decides A in $O(n)$ time

Example (Another decider for $A = \{0^k 1^k \mid k \geq 0\}$)

M''_A = "On input string x:

1. Scan across the tape and if a 0 occurs to the right of a 1, **reject**.
2. Scan across the 0s on tape 1 until the first 1; simultaneously, copy the 0s on 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 we reached the end of the input and all 0s have been crossed off, **accept**; if any 0s remain, **reject**."

Analyzing Algorithms: An Example

- This machine is pretty easy to analyze!

Analyzing Algorithms: An Example

- This machine is pretty easy to analyze!
- Each of the 4 stages uses $O(n)$ steps, so the total running time is $O(n)$

Analyzing Algorithms: An Example

- This machine is pretty easy to analyze!
- Each of the 4 stages uses $O(n)$ steps, so the total running time is $O(n)$
- Note that this is the best possible running time because n steps are necessary at least to read the input!

Analyzing Algorithms: Summary

- Let's recap what we have shown so far

Analyzing Algorithms: Summary

- Let's recap what we have shown so far
- We started from a single-tape TM M_A that decides A in $O(n^2)$ time

Analyzing Algorithms: Summary

- Let's recap what we have shown so far
- We started from a single-tape TM M_A that decides A in $O(n^2)$ time
- Then, we provide an improved version of the TM above, M'_A, which is able to decide A in $O(n \log n)$ time

Analyzing Algorithms: Summary

- Let's recap what we have shown so far
- We started from a single-tape TM M_A that decides A in $O(n^2)$ time
- Then, we provide an improved version of the TM above, M'_A, which is able to decide A in $O(n \log n)$ time
- Finally, we exhibit a two-tape TM M''_A which decides A in $O(n)$ time (i.e., linear time)

Analyzing Algorithms: Summary

- Let's recap what we have shown so far
- We started from a single-tape TM M_A that decides A in $O(n^2)$ time
- Then, we provide an improved version of the TM above, M'_A, which is able to decide A in $O(n \log n)$ time
- Finally, we exhibit a two-tape TM M''_A which decides A in $O(n)$ time (i.e., linear time)
- It turns out that the time complexity of A depends on the characteristics of the model of computation

Complexity vs. Computability

- There is an important difference between complexity and computability theory

Complexity vs. Computability

- There is an important difference between complexity and computability theory
- In computability theory, the Church-Turing thesis guarantees that all *reasonable* models of computation are **equivalent**, i.e., they all decide the same class of languages

Complexity vs. Computability

- There is an important difference between complexity and computability theory
- In computability theory, the Church-Turing thesis guarantees that all *reasonable* models of computation are **equivalent**, i.e., they all decide the same class of languages
- In complexity theory, the choice of the actual model of computation affects the time complexity of languages

Complexity vs. Computability

- There is an important difference between complexity and computability theory
- In computability theory, the Church-Turing thesis guarantees that all *reasonable* models of computation are **equivalent**, i.e., they all decide the same class of languages
- In complexity theory, the choice of the actual model of computation affects the time complexity of languages
- Languages may be decidable in, say, linear time using one model but not necessarily on other models

Table of Contents

Complexity Relationships Among Models

- We examine how the choice of computational model may affect the time complexity of languages

Complexity Relationships Among Models

- We examine how the choice of computational model may affect the time complexity of languages
- To achieve that, we consider **3 models of computation**:
  1. single-tape TM
  2. multi-tape TM
  3. non-deterministic TM

Single- vs. Multi-Tape TM Time Complexity

Theorem

Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ multi-tape TM has an equivalent $O(t^2(n))$ time single-tape TM.

Single- vs. Multi-Tape TM Time Complexity

Theorem

Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ multi-tape TM has an equivalent $O(t^2(n))$ time single-tape TM.

Proof.

We already showed how to convert any multi-tape TM M into a corresponding single-tape TM S that simulates it. The sketch of the proof is to demonstrate that simulating each step of M on S requires at most $O(t(n))$ steps. Hence, the total time used by S is $O(t^2(n))$ □

Single- vs. Multi-Tape TM Time Complexity: Proof

- Let M be a k-tape TM that runs in $t(n)$ time; we must show that a single-tape TM S runs in $O(t^2(n))$ time

Single- vs. Multi-Tape TM Time Complexity: Proof

- Let M be a k-tape TM that runs in $t(n)$ time; we must show that a single-tape TM S runs in $O(t^2(n))$ time
- Machine S uses its single tape to represent the content of all the k tapes of M

Single- vs. Multi-Tape TM Time Complexity: Proof

- Let M be a k-tape TM that runs in $t(n)$ time; we must show that a single-tape TM S runs in $O(t^2(n))$ time
- Machine S uses its single tape to represent the content of all the k tapes of M
- Tapes are stored contiguously, with the position of each k heads of M marked on the appropriate cell

Single- vs. Multi-Tape TM Time Complexity: Proof

- Let M be a k-tape TM that runs in $t(n)$ time; we must show that a single-tape TM S runs in $O(t^2(n))$ time
- Machine S uses its single tape to represent the content of all the k tapes of M
- Tapes are stored contiguously, with the position of each k heads of M marked on the appropriate cell
- Initially, S puts its tape into the format that represents all the k tapes of M, then starts simulating M's steps

Single- vs. Multi-Tape TM Time Complexity: Proof

- To simulate one step of M, S scans all the information stored on its tape to determine the symbols under M's heads

Single- vs. Multi-Tape TM Time Complexity: Proof

- To simulate one step of M, S scans all the information stored on its tape to determine the symbols under M's heads
- Then, S makes another pass over its tape to update its content along with the head positions, according to the transition function of M

Single- vs. Multi-Tape TM Time Complexity: Proof

- To simulate one step of M, S scans all the information stored on its tape to determine the symbols under M's heads
- Then, S makes another pass over its tape to update its content along with the head positions, according to the transition function of M
- If one of the M's heads moves rightward onto the previously unread portion of its tape, S must accommodate for that change by increasing the amount of space reserved for that tape

Single- vs. Multi-Tape TM Time Complexity: Proof

- To simulate one step of M, S scans all the information stored on its tape to determine the symbols under M's heads
- Then, S makes another pass over its tape to update its content along with the head positions, according to the transition function of M
- If one of the M's heads moves rightward onto the previously unread portion of its tape, S must accommodate for that change by increasing the amount of space reserved for that tape
- Basically, S has to shift a portion of its tape to the right

Single- vs. Multi-Tape TM Time Complexity: Proof

- Concerning the actual simulation, for each step of M, S makes two passes over the **active portion** of its tape

Single- vs. Multi-Tape TM Time Complexity: Proof

- Concerning the actual simulation, for each step of M, S makes two passes over the **active portion** of its tape
- The first pass is to obtain the information necessary to determine the next move, the second pass is to carry this out

Single- vs. Multi-Tape TM Time Complexity: Proof

- Concerning the actual simulation, for each step of M, S makes two passes over the **active portion** of its tape
- The first pass is to obtain the information necessary to determine the next move, the second pass is to carry this out
- The length of the active portion of S's tape tells us how long S takes to scan it, so we must determine an upper bound on it

Single- vs. Multi-Tape TM Time Complexity: Proof

- Concerning the actual simulation, for each step of M, S makes two passes over the **active portion** of its tape
- The first pass is to obtain the information necessary to determine the next move, the second pass is to carry this out
- The length of the active portion of S's tape tells us how long S takes to scan it, so we must determine an upper bound on it
- To do so, let's consider the sum of the lengths of the active portions of M's k tapes

Single- vs. Multi-Tape TM Time Complexity: Proof

- Concerning the actual simulation, for each step of M, S makes two passes over the **active portion** of its tape
- The first pass is to obtain the information necessary to determine the next move, the second pass is to carry this out
- The length of the active portion of S's tape tells us how long S takes to scan it, so we must determine an upper bound on it
- To do so, let's consider the sum of the lengths of the active portions of M's k tapes
- Each of these active portions has length at most $t(n)$ because M uses $t(n)$ tape cells in $t(n)$ steps if the head just moves rightward at every single step (**Remember:** M runs in $t(n)$ steps)

Single- vs. Multi-Tape TM Time Complexity: Proof

- Concerning the actual simulation, for each step of M, S makes two passes over the **active portion** of its tape
- The first pass is to obtain the information necessary to determine the next move, the second pass is to carry this out
- The length of the active portion of S's tape tells us how long S takes to scan it, so we must determine an upper bound on it
- To do so, let's consider the sum of the lengths of the active portions of M's k tapes
- Each of these active portions has length at most $t(n)$ because M uses $t(n)$ tape cells in $t(n)$ steps if the head just moves rightward at every single step (**Remember:** M runs in $t(n)$ steps)
- A scan of the active portion of S's tape takes $O(t(n))$ steps

Single- vs. Multi-Tape TM Time Complexity: Proof

- To simulate each of M's steps, S performs two scans and possibly up to k rightward shifts (in the worst case)

Single- vs. Multi-Tape TM Time Complexity: Proof

- To simulate each of M's steps, S performs two scans and possibly up to k rightward shifts (in the worst case)
- Each uses $O(t(n))$ times, so the total time for S to simulate **one** step of M is $O(t(n))$

SAPIENZA
UNIVERSITÀ DI ROMA

Single- vs. Multi-Tape TM Time Complexity: Proof

- To simulate each of M's steps, S performs two scans and possibly up to k rightward shifts (in the worst case)
- Each uses $O(t(n))$ times, so the total time for S to simulate **one** step of M is $O(t(n))$
- To wrap everything up, the initial stage where S lays down the input on the proper format takes $O(n)$ steps

Single- vs. Multi-Tape TM Time Complexity: Proof

- To simulate each of M's steps, S performs two scans and possibly up to k rightward shifts (in the worst case)
- Each uses $O(t(n))$ times, so the total time for S to simulate **one** step of M is $O(t(n))$
- To wrap everything up, the initial stage where S lays down the input on the proper format takes $O(n)$ steps
- Afterwards, S simulates each of the $t(n)$ steps of M using $O(t(n))$ steps, so $t(n) * O(t(n)) = O(t^2(n))$ steps

Single- vs. Multi-Tape TM Time Complexity: Proof

- To simulate each of M's steps, S performs two scans and possibly up to k rightward shifts (in the worst case)
- Each uses $O(t(n))$ times, so the total time for S to simulate **one** step of M is $O(t(n))$
- To wrap everything up, the initial stage where S lays down the input on the proper format takes $O(n)$ steps
- Afterwards, S simulates each of the $t(n)$ steps of M using $O(t(n))$ steps, so $t(n) * O(t(n)) = O(t^2(n))$ steps
- In total, we have $O(n) + O(t^2(n))$ steps; since we have assumed $t(n) \geq n$, the running time of S is $O(t^2(n))$

Deterministic vs. Non-Deterministic TM Time Complexity

- We now consider the analogous theorem yet comparing single-tape vs. non-deterministic TMs

Deterministic vs. Non-Deterministic TM Time Complexity

- We now consider the analogous theorem yet comparing single-tape vs. non-deterministic TMs
- We show that any language that is decidable on the latter machine is also decidable on the former (we already proved that!) yet requiring significantly more time!

Deterministic vs. Non-Deterministic TM Time Complexity

- We now consider the analogous theorem yet comparing single-tape vs. non-deterministic TMs
- We show that any language that is decidable on the latter machine is also decidable on the former (we already proved that!) yet requiring significantly more time!
- Before doing so, let's define the running time of a non-deterministic TM

Deterministic vs. Non-Deterministic TM Time Complexity

- We now consider the analogous theorem yet comparing single-tape vs. non-deterministic TMs
- We show that any language that is decidable on the latter machine is also decidable on the former (we already proved that!) yet requiring significantly more time!
- Before doing so, let's define the running time of a non-deterministic TM
- Remember that a non-deterministic TM is a decider if **all** its computation branches halt on all inputs.

Deterministic vs. Non-Deterministic Time Complexity

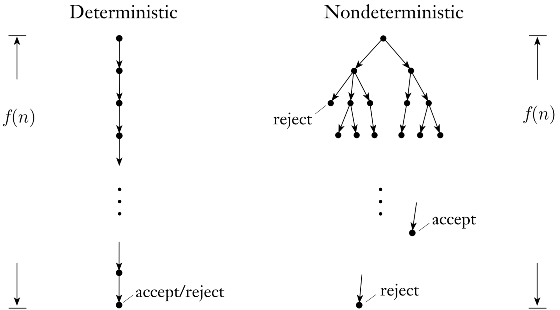
Definition

Let N be a non-deterministic TM that is a decider. The **running time** of N is given by the function $f : \mathbb{N} \mapsto \mathbb{N}$, where $f(n)$ is the **maximum number of steps** that N uses on **any** branch of its computation on any input of length n.

Deterministic vs. Non-Deterministic Time Complexity

Definition

Let N be a non-deterministic TM that is a decider. The **running time** of N is given by the function $f : \mathbb{N} \mapsto \mathbb{N}$, where $f(n)$ is the **maximum number of steps** that N uses on **any** branch of its computation on any input of length n.



Deterministic vs. Non-Deterministic TM Time Complexity

Theorem

Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ non-deterministic TM has an equivalent $2^{O(t(n))}$ time deterministic TM.

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

- Let N be a non-deterministic TM whose running time is $t(n)$

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

- Let N be a non-deterministic TM whose running time is $t(n)$
- We already saw how to construct a deterministic (3-tape) TM D that simulates N by searching through N's non-deterministic computation tree

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

- Let N be a non-deterministic TM whose running time is $t(n)$
- We already saw how to construct a deterministic (3-tape) TM D that simulates N by searching through N's non-deterministic computation tree
- On input of length n, every branch of N's computation tree has length at most $t(n)$

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

- Let N be a non-deterministic TM whose running time is $t(n)$
- We already saw how to construct a deterministic (3-tape) TM D that simulates N by searching through N's non-deterministic computation tree
- On input of length n, every branch of N's computation tree has length at most $t(n)$
- Every node in the tree can have at most b children, where b is the maximum number of legal choices imposed by N's transition function

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

- Let N be a non-deterministic TM whose running time is $t(n)$
- We already saw how to construct a deterministic (3-tape) TM D that simulates N by searching through N's non-deterministic computation tree
- On input of length n, every branch of N's computation tree has length at most $t(n)$
- Every node in the tree can have at most b children, where b is the maximum number of legal choices imposed by N's transition function
- It follows that the total number of leaves of the tree is at most $b^{t(n)}$

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

- The simulation proceeds by exploring this tree **breadth first**

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

- The simulation proceeds by exploring this tree **breadth first**
- In other words, we visit all nodes located at depth d before going to any node at depth $d + 1$

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

- The simulation proceeds by exploring this tree **breadth first**
- In other words, we visit all nodes located at depth d before going to any node at depth $d + 1$
- The total number of nodes in the tree is less than twice the maximum number of leaves, so we bound it by $O(b^{t(n)})$

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

Nodes vs. Leaves

Consider a full b-ary tree of height h, and let $L = b^h$ the **total** number of its leaves.

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

Nodes vs. Leaves

Consider a full b-ary tree of height h, and let $L = b^h$ the **total** number of its leaves.
Let $N = \sum_{i=0}^{h} b^i = b^0 + b^1 + b^2 + \ldots b^h$ the **total** number of nodes.

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

Nodes vs. Leaves

Consider a full b-ary tree of height h, and let $L = b^h$ the **total** number of its leaves.
Let $N = \sum_{i=0}^{h} b^i = b^0 + b^1 + b^2 + \ldots b^h$ the **total** number of nodes.
The expression above is a finite geometric series $\sum_{i=0}^{h} ar^i$, where $a = 1$ and $r = b$, whose closed-form solution is:

$$N = \frac{1 - b^{h+1}}{1 - b} = \frac{1 - (b \cdot b^h)}{1 - b} = \frac{1 - bL}{1 - b} < 2L = O(L) = O(b^h)$$

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

Nodes vs. Leaves

$$N = \frac{1 - b^{h+1}}{1 - b} = \frac{1 - (b \cdot b^h)}{1 - b} = \frac{1 - bL}{1 - b} < 2L = O(L) = O(b^h)$$

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

Nodes vs. Leaves

$$N = \frac{1-b^{h+1}}{1-b} = \frac{1-(b \cdot b^h)}{1-b} = \frac{1-bL}{1-b} < 2L = O(L) = O(b^h)$$

As $b \geq 2$, we have:

$$N = \frac{1-2L}{1-2} = 2L - 1 \quad (b = 2);$$
$$N = \frac{1-3L}{1-3} = 3/2L - 1/2 \quad (b = 3);$$
$$N = \frac{1-4L}{1-4} = 4/3L - 1/3 \quad (b = 4);$$
$$\ldots$$

At worst, in a binary tree (i.e., when $b = 2$), the total number of nodes is **less than twice** the number of leaves.

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

- The time for starting from the root and traveling down to a node is $O(t(n))$

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

- The time for starting from the root and traveling down to a node is $O(t(n))$
- Therefore, the running time of D is $O(t(n)b^{t(n)})$

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

- The time for starting from the root and traveling down to a node is $O(t(n))$
- Therefore, the running time of D is $O(t(n)b^{t(n)})$
- Notice that $b^x = (\underbrace{c^{\log_c b}}_{b})^x$; if we set $x = t(n)$ and $c = 2$, we obtain:

$$b^{t(n)} = (2^{\log_2 b})^{t(n)} = 2^{\log_2 b * t(n)} = 2^{k * t(n)}, \text{ where } k = \log_2 b$$

$$b^{t(n)} = 2^{O(t(n))}$$

Deterministic vs. Non-Deterministic TM Time Complexity: Proof

- The time for starting from the root and traveling down to a node is $O(t(n))$
- Therefore, the running time of D is $O(t(n)b^{t(n)})$
- Notice that $b^x = (\underbrace{c^{\log_c b}}_{b})^x$; if we set $x = t(n)$ and $c = 2$, we obtain:

$$b^{t(n)} = (2^{\log_2 b})^{t(n)} = 2^{\log_2 b * t(n)} = 2^{k*t(n)}, \text{ where } k = \log_2 b$$

$$b^{t(n)} = 2^{O(t(n))}$$

- Overall, the running time complexity of D is $O(t(n)) * 2^{O(t(n))} = 2^{O(t(n))}$

Deterministic vs. Non-Deterministic TM Time Complexity

- The result above is obtained assuming D is a 3-tape TM

Deterministic vs. Non-Deterministic TM Time Complexity

- The result above is obtained assuming D is a 3-tape TM
- Interestingly enough, D would have the same time complexity even if it was a single-tape TM

Deterministic vs. Non-Deterministic TM Time Complexity

- The result above is obtained assuming D is a 3-tape TM
- Interestingly enough, D would have the same time complexity even if it was a single-tape TM
- This is because a conversion from a multi-tape TM to a single-tape TM at most squares the running time

Deterministic vs. Non-Deterministic TM Time Complexity

- The result above is obtained assuming D is a 3-tape TM
- Interestingly enough, D would have the same time complexity even if it was a single-tape TM
- This is because a conversion from a multi-tape TM to a single-tape TM at most squares the running time

$$(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$$

Complexity Relationships: Final Remarks

- Polynomial differences in running time are considered to be small, whereas exponential differences are considered to be large: this is because of their respective growth rates

Complexity Relationships: Final Remarks

- Polynomial differences in running time are considered to be small, whereas exponential differences are considered to be large: this is because of their respective growth rates
- It is for this reason that, for larger problems and larger inputs, the difference between using a single-tape TM and a multi-tape TM is negligible

Complexity Relationships: Final Remarks

- Polynomial differences in running time are considered to be small, whereas exponential differences are considered to be large: this is because of their respective growth rates
- It is for this reason that, for larger problems and larger inputs, the difference between using a single-tape TM and a multi-tape TM is negligible
- All reasonable deterministic computational models are **polynomially equivalent**; this means that any one of them can simulate another with only a polynomial increase in running time