

7. Time Complexity

- Before we have not paid any attention to the time required to solve a problem
- Now we turn to consider solvable problems and their time complexity
- A problem that in principle is solvable can in practice require so much time that it effectively is unsolvable
- In TSP (traveling salesperson problem) the problem is to find the shortest *Hamilton cycle* from a weighted graph
- Trivial algorithm: in a graph of n nodes, go through all possible $n!$ routes and choose the shortest one



- In a computer in which examining one route takes 0.001 s, the trivial algorithm would require more time than the current age of the universe for a graph of $n = 22$ nodes
- If we had a billion times more efficient computer then we still couldn't solve the TSP for a graph of $n = 31$ nodes within the current age of the universe
- Hence, it is not reasonable to call the exponential ($n! \approx O(n^n)$) algorithm that goes through all the potential routes as a solution for the TSP
- One does not know an algorithm whose time consumption would be bounded by a polynomial of n ; on the other hand, one is not able to show that such an algorithm does not exist



7.1 Measuring Complexity

- The **length** of the computation of a standard Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \text{accept}, \text{reject})$

$$q_0 w \Rightarrow_M^* u q a v, \quad q \in \{ \text{accept}, \text{reject} \}$$

is the number of steps included

- Time complexity on input w :

$$\text{time}_M = \begin{cases} \text{length of } q_0 w \xrightarrow{*} \dots, & \text{if the computation halts} \\ \infty, & \text{if computation doesn't halt on } w \end{cases}$$

- The **running time** or **time complexity** of M is the function

$$\text{time}_M: \Sigma^* \rightarrow \mathbb{N} \cup \{ \infty \}$$



- One usually examines time complexity as a function of the length n of the input:

- In the average case* when inputs of length n are drawn from the probability distribution $P_n(w)$

$$\text{time}_M^{\text{avg}}(n) = \sum_{|w|=n} P_n(w) \cdot \text{time}_M(w)$$

- Or more commonly *in the worst case*

$$\text{time}_M^{\text{max}}(n) = \max_{|w|=n} \text{time}_M(w)$$

- Usually the notation is also simplified

$$\text{time}_M: \mathbb{N} \rightarrow \mathbb{N} \cup \{ \infty \},$$

$$\text{time}_M(n) = \text{time}_M^{\text{max}}(n)$$

- It would be interesting to analyze the average case, but it is typically so difficult that one has to be content with dealing with the worst-case analysis



Asymptotic Analysis

- Even the worst-case running times are still too complex for exact examination
- Therefore, it is usual to just estimate their growth rates by **asymptotic notation**
- Let $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$ be arbitrary functions
 - $f = O(g)$: g is an (asymptotic) *upper bound* for f , if $\exists c, n_0 \in \mathbb{N}$:

$$f(n) \leq c \cdot g(n), \quad \forall n \geq n_0$$
 - $f = \Theta(g)$: g is an *asymptotically tight bound* for f , if $f = O(g)$ and $g = O(f)$



- $f = o(g)$: g is an *asymptotically tight upper bound* for f , if $\forall c > 0: \exists n_c \in \mathbb{N}$:

$$f(n) < c \cdot g(n), \quad \forall n \geq n_c$$
- $f = \Omega(g)$: g is an *asymptotic lower bound* for f , if $\exists c > 0$: for infinitely many $n \in \mathbb{N}$:

$$f(n) \geq c \cdot g(n)$$
- $f = o(g) \Leftrightarrow f = O(g) \wedge g \neq O(f)$
 $f \neq \Omega(g) \Leftrightarrow f = o(g)$
- One usually talks more vaguely and says, e.g.,
 - "function $n!$ has growth rate $O(n^n)$ " or
 - " $2n^2$ has growth rate n^2 "





Lemma N

1. $\log_a n = \Theta(\log_b n) \quad \forall a, b > 0,$
2. $n^a = o(n^b),$ if $a < b,$
3. $2^{an} = o(2^{bn}),$ if $a < b,$
4. $\log_a n = o(n^b) \quad \forall a, b > 0,$
5. $n^a = o(2^{bn}) \quad \forall a, b > 0,$
6. $c \cdot f(n) = \Theta(f(n)) \quad \forall c > 0$ and functions $f,$
7. $f(n) + g(n) = \Theta(\max\{f(n), g(n)\})$ for all functions f and $g,$ and
8. if $p(n)$ is a polynomial of degree $r,$ then $p(n) = \Theta(n^r)$

□



Lemma O Let $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$ be arbitrary functions. If the limit

$$L = \lim_{n \rightarrow \infty} f(n)/g(n)$$

exists and

1. $0 < L < \infty \Rightarrow f = \Theta(g)$
2. $L = 0 \Rightarrow f = o(g)$
3. $L > \infty \Rightarrow g = o(f)$
4. $L < \infty \Rightarrow f = O(g)$
5. $L > 0 \Rightarrow f = \Omega(g)$

□





- Let us examine the recognition of the familiar, non-regular language $A = \{0^k1^k \mid k \geq 0\}$
- How much time does a single-tape Turing machine require in deciding A ?
- 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 0 s and 1 s remain on the tape:
 3. Scan across the tape, crossing off a single 0 and a single 1
 4. If 0 s still remain after all the 1 s have been crossed off, or if 1 s still remain after all the 0 s have been crossed off, *reject*. Otherwise, if neither 0 s nor 1 s remain on the tape, *accept*.



- In stage 1 of the recognizer for A uses n steps on input w , $|w| = n$
- Repositioning the head at the left-hand end of the tape uses another n steps
- In total stage 1 uses $2n = O(n)$ steps
- In stages 2 and 3 crossing off a pair of 0 s and 1 s requires $O(n)$ steps and at most $n/2$ such scans can occur
- Altogether, stages 2 and 3 take $O(n^2)$ steps
- Stage 4 again takes a linear number of steps
- All in all the Turing machine requires time

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





- Recognizing A , however, is not an $\Omega(n^2)$ task; the following Turing machine only takes $O(n \log n)$ time
- 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 0 s and 1 s remain on the tape:
 - a) If the total number of 0 s and 1 s remaining is odd, *reject*.
 - b) Cross off every other 0 starting with the first 0 . Do the same with the 1 s
 3. If no 0 s and no 1 s remain on the tape, *accept*. Otherwise, *reject*



- On every scan performed in stage 2, the total number of 0 s remaining is cut in half and any remainder is discarded
- Every stage takes $O(n)$ time and there are at most $1 + \log_2 n$ iterations of stage 2
- The total time is $O(n \cdot \log n)$
- Examples of the correctness of the TM: if there initially were, e.g., seven 0 s and six 1 s, then the parity test immediately rejects
- If seven 0 s and five 1 s, then after the first halving of 0 s and 1 s there remains three 0 s and two 1 s and the parity test rejects
- If seven 0 s and three 1 s, then it takes two rounds of halving before the parity test gets to reject



- A single-tape Turing machine cannot decide the language A in less asymptotic time than $O(n \cdot \log n)$
- Any language that can be decided in $o(n \cdot \log n)$ time on a single-tape TM is regular
- If, on the other hand, we have a second tape, we can decide the language in linear time
- On input string w :
 1. if a 0 is found to the right of a 1 , then *reject*
 2. Copy the 0 s onto tape 2
 3. Scan across the 1 s, for each cross off a 0 on tape 2. If all 0 s are crossed off before all the 1 s are read, *reject*
 4. If all 0 s have now been crossed off, *accept*. If any 0 s remain, *reject*.



- Universal models of computation are thus not equally efficient
- In complexity theory it does matter which is the model being used, while in computability theory they are all equivalent
- Time requirements for deterministic models do not differ greatly
- Let $t: \mathbb{N} \rightarrow \mathbb{R}^+$ be an arbitrary function
- A language A can be decided in time t , if there exists a *deterministic* Turing machine M s.t.
 - $L(M) = A$ and
 - $\text{time}_M(n) \leq t(n)$ for all n
- The time complexity class for formal languages

$$\text{DTIME}(t) = \{ A \mid A \text{ can be decided in time } t \}$$





- Every multitape Turing machine has an equivalent single-tape Turing machine (Theorem 3.13)
- Simulating each step of the k -tape machine uses at most $O(t(n))$ steps on the single-tape machine, where $t(n) \geq n$ is the time complexity of the multitape machine
- There are in total $O(t(n))$ steps, and hence

Theorem 7.8 Every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

- The running time of a nondeterministic Turing machine N : $\text{time}_N(w)$ is the length of the *longest* computation ($q_0 w \Rightarrow_N \dots$)



- The computations of a nondeterministic decider on input w may be thought as a computation tree.
- In the leaves of the tree the computation halts. The TM accepts the input w , if some leaf corresponds to the accept state.
- The worst-case time complexity:

$$\text{time}_N(n) = \max_{|w|=n} \text{time}_N(w)$$

- Language A can be decided nondeterministically in time t , if there exists a nondeterministic decider N s.t.
 - $L(N) = A$ and
 - $\text{time}_N(n) \leq t(n) \quad \forall n$
- Nondeterministic time complexity classes (Definition 7.21):
 $\text{NTIME}(t) = \{ A \mid A \text{ can be decided nondeterministically in time } t \}$





Theorem 7.11 Every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

Proof. The deterministic Turing machine M of the proof of Theorem 3.16 systematically searches the computation tree of the nondeterministic machine N .

Let the running time of N be $t(n)$. Then, on input of length n every branch of N 's nondeterministic computation tree has length at most $t(n)$.

The branching factor of the tree is determined by the transition possibilities in the transition function of N . Let $b \geq 2$ its upper bound.

The number of leaves in the tree is at most $b^{t(n)}$.



In the worst case we have to examine all the nodes in the tree. The total number of nodes in the tree is at most less than twice that of the leaves; i.e., of the order $O(b^{t(n)})$. Hence, the time required by the three-tape TM M is asymptotically $2^{O(t(n))}$.

By Theorem 7.8 converting to a single-tape TM at most squares the running time. Thus, the running time of the single-tape simulator is $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$. \square

- The efficiency difference between a single-tape and a multitape TM is at most the square of t ; i.e., polynomial in t
- On the other hand, the efficiency difference between a deterministic and a nondeterministic TM may be exponential in t



7.1 The Class P

- For our purposes, polynomial differences in running time are considered to be small, whereas exponential differences are considered to be large
- In time complexity polynomial time requirement is thought as useful, whereas exponential time requirement most often is useless
- Exponential time algorithms typically arise in exhaustive searching (brute-force search)
- All reasonable models of computation are polynomially equivalent
- Any one of them can simulate another with only a polynomial increase in running time



• **Definition 7.12** $P = \bigcup \{ \text{DTIME}(t) \mid t \text{ is a polynomial} \}$
 $= \bigcup_{k \geq 0} \text{DTIME}(n^k)$

- The class **P** plays a central role and is important because
 - It is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and
 - It roughly corresponds to the class of problems that are realistically solvable on a computer
- Algorithms with a high-degree polynomial time complexity of course are not that practical, but they tend to be quite rare
- The role of **P** in complexity theory is similar to the role of decidable languages in computability theory



Examples of Problems in P

- Let G be a directed graph containing nodes s and t
- Problem PATH: Is there a directed path in G from s to t ?
- Brute-force algorithm would examine all potential paths in G to determine whether any is the required path
- In the worst case the number of potential paths is exponential in the number of nodes of the graph
- Thus we need a more sophisticated algorithm:
 1. Place a mark on node s
 2. As long as the set of marked nodes keeps growing:
Scan all the edges of G . If an edge (a, b) goes from a marked node a to an unmarked node b , mark node b
 3. If t is marked, *accept*. Otherwise, *reject*.



- Let m be the number of nodes in graph G
- Stages 1 and 3 are executed only once
- Stage 2 runs at most m times, because each time except the last it marks an additional node in G
- Thus, the total number of stages used is $m + 2$
- Each stage is easy to implement in polynomial time

Theorem 7.14 $\text{PATH} \in \text{P}$.





- Say that two numbers are *relatively prime* if 1 is the largest integer that evenly divides them both
- For example, 10 and 21 are relatively prime, even though neither of them is a prime number by itself
- On the other hand, 9 and 21 are both divisible by 3
- Let **RP** be the problem of testing whether two numbers are relatively prime

$$\text{RP} = \{ \langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime} \}$$

- Again, brute-force algorithm is inefficient
- Searching through all possible divisors of both numbers requires exponential time, because the number of possible divisors is exponential in the length of the binary representation of the given numbers x and y



RP is in P

- An ancient numerical procedure, called the **Euclidean algorithm**, for computing the *greatest common divisor* (**gcd**) helps us to solve **RP** efficiently
- For example, $\text{gcd}(18, 24) = 6$
- Obviously, x and y are relatively prime iff $\text{gcd}(x, y) = 1$
- Let x and y be natural numbers in binary representation
 1. Repeat until $y = 0$
 - a) $x \leftarrow x \bmod y$
 - b) exchange x and y
 2. Output x





- E.g., $x = 18, y = 24$
 - $x \leftarrow 18 \bmod 24 = 18$
 - $x \leftarrow 24, y \leftarrow 18$
 - $x \leftarrow 24 \bmod 18 = 6$
 - $x \leftarrow 18, y \leftarrow 6$
 - $x \leftarrow 18 \bmod 6 = 0$
 - $x \leftarrow 6, y \leftarrow 0$
- Output $x = 6$



- After the stage 1 has been executed for the first time, it definitely holds that $x > y$
- Thereafter, every execution of stage 1(a) cuts the value of x by at least half:
 - $x \geq 2y \Leftrightarrow x \bmod y < y \leq x/2$
 - $x < 2y \Leftrightarrow x \bmod y = x - y < x/2$
- Every other loop through of stage1 reduces each of the original values x and y by at least half, so in the worst case the number of loops is at most

$$\min\{2\log_2 x, 2\log_2 y\}$$
- The binary representations of the numbers have logarithmic length, and hence the Euclidean algorithm requires a linear time