Chapter 14

Time Complexity

Time Complexity

- The study of *complexity* of *a* <u>problem</u> is the study of the *complexity* of *the* <u>algorithm</u> that solves the problem.

- The computational complexity of an algorithm is measured by the amount of *resources* required to carry it out, i.e., *time* and *space*.

- The <u>time</u> <u>complexity</u> of a computation *C* is determined by the amount of time required to perform *C*, whereas the <u>space</u> <u>complexity</u> of *C* is determined by the amount of *storage space* required by *C*.

14.2 Rates of Growth

- The rate of growth of a function, which measures the increase of the function values as the input gets *arbitrarily large*, is determined by the most significant contributor to the growth of the function.

- Table 14.2 (*Growth of functions*)

n	0	5	10	25	50	100	1,000
$20n + 500$	500	600	700	1,000	1,500	2,500	20,500
n^2	0	25	100	625	2,500	10,000	1,000,000
$n^2 + 2n + 5$	5	40	125	680	2,605	10,205	1,002,005
$n^2 /(n^2 + 2n + 5)$	0	0.625	0.800	0.919	0.960	0.980	0.998

  - the linear and constant terms of $n^2 + 2n + 5$ are called the lower-order terms, which do *not* significantly contribute to the growth of the function values.

14.2 Rates of Growth

- Defn 14.2.1 Let $f : N \to N$ and $g : N \to N$ be one-variable number-theoretic functions.

  i. f is said to be of order g, written $f \in O(g)$, if there is a positive constant c and natural number n_0 such that

  $$f(n) \le c \times g(n),\ \forall n \ge n_0$$

  ii. $O(g) = \{ f \mid f \text{ is of order } g \}$, i.e., the set of all functions of order g, is called the "big oh of g"

  - f is of order g, written $f \in O(g)$, if the growth of f is bounded by a *constant* multiple of the *values* of g

14.2 Rates of Growth

- The rate of growth is determined by the most significant contributor to the growth of the function.

- If $f \in O(g)$ and $g \in O(f)$, then given two positive constants C_1 and C_2,

$$f(n) \le C_1 \times g(n),\ \forall n \ge n_1;$$
$$g(n) \le C_2 \times f(n),\ \forall n \ge n_2$$

  - f and g have the *same* rate of growth, *i.e*., neither f nor g grow faster than the other.

14.2 Rates of Growth

- A function f is said to <u>exponentially</u> (<u>polynomially</u>, respectively) *bounded* if $f \in O(2^n)$ ($f \in O(n^r)$, respectively).

  - <u>Example 14.2.2</u> Let $f(n) = n^2 + 2n + 5$ and $g(n) = n^2$. Then

    $g \in O(f)$, since $n^2 \leq n^2 + 2n + 5$, $\forall_{n \in N}$, $n_0 = 0$, and $C = 1$.

    $f \in O(g)$, since $2n \leq 2n^2$ and $5 \leq 5n^2$, $\forall_{n \geq 1}$.

    Then $f(n) = n^2 + 2n + 5 \leq n^2 + 2n^2 + 5n^2 = 8n^2 = 8 \times g(n)$, $\forall_{n \geq 1}$.

    Thus $f \in O(n^2)$

  - <u>Example 14.2.1</u> Let $f(n) = n^2$ and $g(n) = n^3$. $f \in O(g)$, but $g \notin O(f)$.

    Clearly, $n^2 \in O(n^3)$, since $n^2 \leq n^3$, for $\forall_{n \in N}$, $n_0 = 0$, and $C = 1$.

    Suppose that $n^3 \in O(n^2)$. Then there exists constants C and n_0 .∍.

    $$n^3 \leq C \times n^2, \ \forall_{n \geq n_0}.$$

    Let $n_1 = \max(n_0 + 1, C + 1)$. Then

    $$n_1^3 = n_1 \times n_1^2 > C \times n_1^2, \text{ since } n_1 > C,$$

    contradicting the inequality that $n_1^3 \leq C \times n_1^2$. Thus $n^3 \notin O(n^2)$.

14.2 Rates of Growth

- Using *limits* to determine the asymptotic complexity of two functions
- Let f & g be two number-theoretic functions
  - If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f \in O(g)$, but $g \notin O(f)$
  - If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ with $0 < c < \infty$, then $f \in \Theta(g)$, and $g \in \Theta(f)$

    where $\Theta(g) = \{ f \mid f \in O(g) \text{ and } g \in O(f) \}$
  - If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, then $f \notin O(g)$ and $g \in O(f)$
- Example. Let $f(n) = n^2$ and $g(n) = n^3$. $f \in O(g)$, but $g \notin O(f)$.

  Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{1}{n} = 0$, $f \in O(g)$, but $g \notin O(f)$.

14.2 Rates of Growth

TABLE 14.3 A Big O Hierarchy

Big Oh (Big O)	**Name**
$O(1)$	constant
$O(\log_a(n))$	logarithmic
$O(n)$	linear
$O(n \log_a(n))$	$n \log n$
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^r)$	polynomial $r \geq 0$
$O(b^n)$	exponential $b > 1$
$O(n!)$	factorial

14.2 Rates of Growth

TABLE 14.4 Number of Transitions of Machine with Time Complexity tc_M with Input of Length n

n	$\log_2(n)$	n	n^2	n^3	2^n	$n!$
5	2	5	25	125	32	120
10	3	10	100	1,000	1,024	3,628,800
20	4	20	400	8,000	1048576	2.4×10^{18}
30	4	30	900	27,000	1.0×10^{9}	2.6×10^{32}
40	5	40	1,600	64,000	1.1×10^{12}	8.1×10^{47}
50	5	50	2,500	125,000	1.1×10^{15}	3.0×10^{64}
100	6	100	10,000	1,000,000	1.2×10^{30}	$> 10^{157}$
200	7	200	40,000	8,000,000	1.6×10^{60}	$> 10^{374}$

14.3 Time Complexity

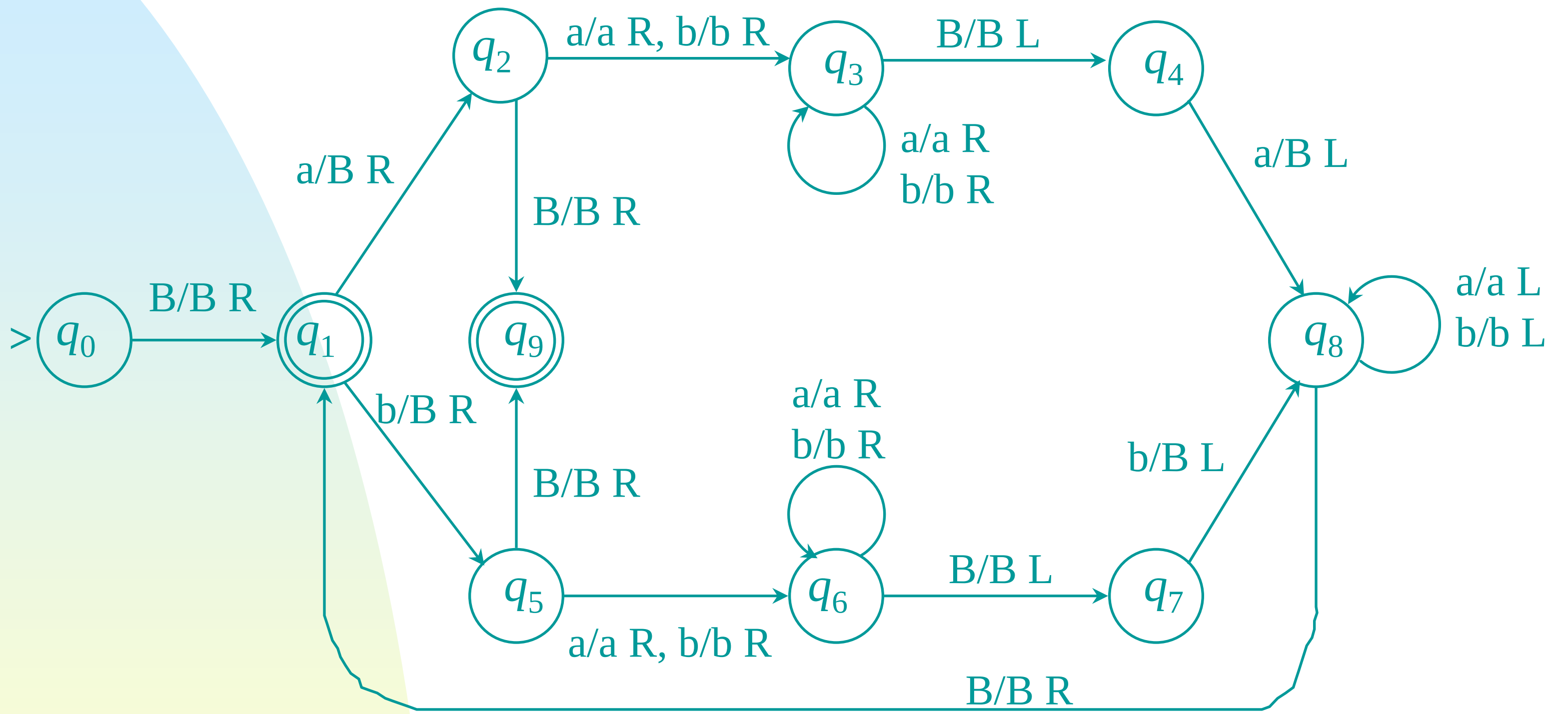
- Computational Complexity of TMs
  - Given a TM M, M accepts the language L in *polynomial-time* if $L = L(M)$ and there exists $P(n)$, a polynomial expression, such that M accepts any $w \in L(M)$ in $\leq P(|w|)$ steps.
  - A problem is <u>tractable</u> if there is a DTM that solves the problem whose computations are *polynomially bounded*.
  - A language L is <u>decidable</u> in *polynomial time* if there exists a TM M that accepts L with $tc_M \in O(n^r)$, where $r \in N$ is independent of n.
  - The family of languages decidable in polynomial time is denoted P, e.g.,

    Q: Is the language of palindromes over $\{a, b\}$ *decidable* in *polynomial time*?

    A: yes, there exists TM A that accepts the language in $O(n^2)$ time as shown on P.444 (a TM that accepts palindromes), where $tc_M(n) = (n^2 + 3n + 2) / 2 \in O(n^2)$
  - A language L is said to be accepted in *non-deterministic polynomial time* if there is a NDTM M that accepts L with $tc_M \in O(n^r)$, $r \in N$ and r is independent of n. This family of language is denoted NP.

14.3 Time Complexity of a TM

- Example: Let the transition diagram of the TM M that accepts the language of palindromes over $\{a, b\}$ be



(* When evaluating the time complexity of a TM, we assume that the computations terminates for every input since it makes no sense to discuss the efficiency of a computation that continue *indefinitely* *)

Consider Table 14.5. Hence, $tc_M(0) = 1$; $tc_M(1) = 3$; $tc_M(2) = 6$; $tc_M(3) = 10$

14.3 Time Complexity of a TM

- Example. Consider the actions of *M* when processing an *even*-length (palindrome) input string. The computation alternates between sequences of right (RM) and left (LM) movements.
  - RM: requires k+1 transitions, where k is the length of the *non-blank* portion of the tape
  - LM: this requires k transitions
  - A pair of RM and LM reduces the length of nonblank portion of the tape by 2.
  - It requires n/2 iterations of the RM-LM for accepting a (palindrome) string of $|n|$

Iteration	Direction	Transitions
1	R	n+1
	L	n
2	R	n-1
	L	n-2
3	R	n-3
	L	n-4
:	:	:
(n/2) + 1	R	1

The time complexity of *M* is (same as *odd*-length): $\sum_{i=1}^{n+1} i = (n+2)(n+1)/2 \in O(n^2)$

14.3 Time Complexity of a TM

- Time complexity of a TM is determined by the *maximum number of steps* (i.e., transitions) executed during the computation, whereas the space complexity of a TM computation is defined to be *number of tape cells* required by the computation.

- If the time complexity of a TM computation is n, then the space complexity of that computation is $\leq n + 1$.

  - The *space complexity* of a TM computation may be *less than the number of cells* to hold the processing string as the TM may not have to process the entire string in order to accept it.

- We measure the *time complexity* of a TM M by the *worst-case* performance of M (occurred when M accepts the input string) with input string of length n, denoted $tc_M(n)$.

14.3 Analysis of the complexity of TMs

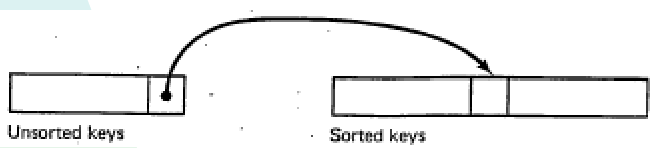
- Example 14.3.1. Given a two-tape TM *M*' that accepts the set of palindromes over { *a*, *b* }, $tc_{M'}(n) = 3(n + 1) + 1$.
  - there is a tradeoff between the complexity of a transition and the number of steps between *M* (in previous example) and *M*'

- Complexity of Algorithms
  - The *time* required to execute an algorithm tends to be a function of the *length of the input*.
  - Average-case complexity: computed by first multiplying the complexity of each possible computation (C_i) by the probability of that computation occurring (P_i) and then adding these product, i.e., $\sum_{i=1}^{m} P_i \cdot C_i$
  - The analysis requires the identification of the *dominating steps* in the algorithm and estimates the number *of times* these steps will be performed.

14.4 Analysis of the complexity of TMs

- Complexity of Algorithms
  - <u>Example</u> (<u>Insertion Sort Algorithm</u>). The time complexity of the algorithm is proportional to the *number of times* the body of the *WHILE*-loop is executed.

AN INSERTION SORT



Consider again our example unsorted list of keys:

14 3 22 9 10 14 2 7 25 6

The first pass considers the first key, which is 14, and results in:

3 22 9 10 14 2 7 25 6 — unsorted list
14 — sorted list

After the second pass:

22 9 10 14 2 7 25 9 — unsorted list
3 14 — sorted list

After the sixth pass:

2 7 25 6 — unsorted list
3 9 10 14 14 22 — sorted list

Sorted List: 1) 14 (3) pivot
2) 3 14 (22) pivot
3) 3 14 22 (9) pivot
4) 3 9 14 22

Insertion Sort Algorithm

```
Progam InsertSort(inout: List; in: ListLength)
var    PivotPosition, I: integer;
       Pivot: ListEntryType;
begin
  if (ListLength ≥ 2) then
       begin
          PivotPosition := 2;
          repeat
               Pivot := List[PivotPosition];
               I := PivotPosition;
               while (I > 1 and List[I - 1] > Pivot) do
                 begin
                       List[I] := List[I-1];
                       I := I - 1;
                 end;
               List[I] := Pivot;
               PivotPosition := PivotPosition + 1;
          until (PivotPosition > ListLength)
       end.
```

14.4 Analysis of the complexity of TMs

- Example (Insertion Sort Algorithm) (Continued).

  Worst-case: when the original list is in the reverse order, since

  When the pivot's entry is the 2nd entry, the loop body is executed 1

  When the pivot's entry is the 3rd entry, the loop body is executed 2

  :

  When the pivot's entry is the m^{th} entry, the loop body is executed $m - 1$

  Hence if |list| = n, the worst case complexity of the algorithm is

  $$1 + 2 + \ldots + (n - 1) = n(n - 1)/2 = 1/2(n^2 - n) \in O(n^2)$$