Time complexity

From Wikipedia, the free encyclopedia

"Running time" redirects here. For the film, see Running Time (film).

In computer science, the **time complexity** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input[1]:226. The time complexity of an algorithm is commonly expressed using big O notation, which excludes coefficients and lower order terms. When expressed this way, the time complexity is said to be described *asymptotically*, i.e., as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$ for any n (bigger than some n_0), the asymptotic time complexity is $O(n^3)$.

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

Since an algorithm's performance time may vary with different inputs of the same size, one commonly uses the worst-case time complexity of an algorithm, denoted as $\boldsymbol{T(n)}$, which is defined as the maximum amount of time taken on any input of size n. Less common, and usually specified explicitly, is the measure of average-case complexity. Time complexities are classified by the nature of the function $T(n)$. For instance, an algorithm with $T(n) = O(n)$ is called a *linear time algorithm*, and an algorithm with $T(n) = O(M^n)$ and $m^n = O(T(n))$ for some $M \geq m > 1$ is said to be an *exponential time algorithm*.

Contents

Table of common time complexities

Further information: Computational complexity of mathematical operations

The following table summarizes some classes of commonly encountered time complexities. In the table, poly(x) = $x^{O(1)}$, i.e., polynomial in x.

Name	Complexity class	Running time ($T(n)$)	Examples of running times	Example algorithms
constant time		$O(1)$	10	Determining if an integer (represented in binary) is even or odd
inverse Ackermann time		$O(\alpha(n))$		Amortized time per operation using a disjoint set
iterated logarithmic time		$O(\log^* n)$		Distributed coloring of cycles
log-logarithmic		$O(\log \log n)$		Amortized time per operation using a bounded priority queue[2]
logarithmic time	DLOGTIME	$O(\log n)$	$\log n$, $\log(n^2)$	Binary search
polylogarithmic time		$\text{poly}(\log n)$	$(\log n)^2$	
fractional power		$O(n^c)$ where $0 < c < 1$	$n^{1/2}$, $n^{2/3}$	Searching in a kd-tree
linear time		$O(n)$	n	Finding the smallest or largest item in an unsorted array
"n log star n" time		$O(n \log^* n)$		Seidel's polygon triangulation algorithm.
linearithmic time		$O(n \log n)$	$n \log n$, $\log n!$	Fastest possible comparison sort
quadratic time		$O(n^2)$	n^2	Bubble sort; Insertion sort; Direct convolution
cubic time		$O(n^3)$	n^3	Naive multiplication of two $n\times n$ matrices. Calculating partial correlation.
polynomial time	P	$2^{O(\log n)} = \text{poly}(n)$	n, $n \log n$, n^{10}	Karmarkar's algorithm for linear programming; AKS primality test
quasi-polynomial time	QP	$2^{\text{poly}(\log n)}$	$n^{\log \log n}$, $n^{\log n}$	Best-known $O(\log^2 n)$-approximation algorithm for the directed Steiner tree problem.
sub-exponential time (first definition)	SUBEXP	$O(2^{n^{\varepsilon}})$ for all $\varepsilon > 0$	$O(2^{\log n^{\log \log n}})$	Assuming complexity theoretic conjectures, BPP is contained in SUBEXP.[3]
sub-exponential time (second definition)		$2^{o(n)}$	$2^{n^{1/3}}$	Best-known algorithm for integer factorization and graph isomorphism
exponential time (with linear	E	$2^{O(n)}$	1.1^n, 10^n	Solving the traveling salesman problem using dynamic programming

exponent)				
exponential time	EXPTIME	$2^{\text{poly}(n)}$	2^n, 2^{n^2}	Solving matrix chain multiplication via brute-force search
factorial time		$O(n!)$	$n!$	Solving the traveling salesman problem via brute-force search
double exponential time	2-EXPTIME	$2^{2^{\text{poly}(n)}}$	2^{2^n}	Deciding the truth of a given statement in Presburger arithmetic

Constant time

An algorithm is said to be **constant time** (also written as **O(1)** time) if the value of $T(n)$ is bounded by a value that does not depend on the size of the input. For example, accessing any single element in an array takes constant time as only one operation has to be performed to locate it. However, finding the minimal value in an unordered array is not a constant time operation as a scan over each element in the array is needed in order to determine the minimal value. Hence it is a linear time operation, taking O(n) time. If the number of elements is known in advance and does not change, however, such an algorithm can still be said to run in constant time.

Despite the name "constant time", the running time does not have to be independent of the problem size, but an upper bound for the running time has to be bounded independently of the problem size. For example, the task "exchange the values of *a* and *b* if necessary so that $a \le b$" is called constant time even though the time may depend on whether or not it is already true that $a \le b$. However, there is some constant *t* such that the time required is always *at most t*.

Here are some examples of code fragments that run in constant time:

```
int index = 5;
int item = list[index];
if (condition true) then
   perform some operation that runs in constant time
else
   perform some other operation that runs in constant time
for i = 1 to 100
   for j = 1 to 200
      perform some operation that runs in constant time
```

If $T(n)$ is O(*any constant value*), this is equivalent to and stated in standard notation as $T(n)$ being O(1).

Logarithmic time

Further information: Logarithmic growth

An algorithm is said to take **logarithmic time** if $T(n)$ = **O(log *n*)**. Due to the use of the binary numeral system by computers, the logarithm is frequently base 2 (that is, $\log_2 n$, sometimes written lg *n*). However, by the change of base for logarithms, $\log_a n$ and $\log_b n$ differ only by a constant multiplier, which in big-O notation is discarded; thus O(log *n*) is the standard notation for logarithmic time algorithms regardless of the base of the logarithm.

Algorithms taking logarithmic time are commonly found in operations on binary trees or when using binary search.

An O(log n) algorithm is considered highly efficient, as the operations per instance required to complete decrease with each instance.

A very simple example of this type is an algorithm that cuts a string in half, then cuts the right half in half, and so on. It will take O(log n) time (n being the length of the string) since we chop the string in half before each print (we make the assumption that *console.log* and *str.substring* run in constant time). This means, in order to increase the number of prints, we have to double the length of the string.

```
// Function to recursively print the right half of a string
var right = function(str){
    var length = str.length;

    // Helper function
    var help = function(index){

        // Recursive Case: Print right half
        if(index < length){

            // Prints characters from index until the end of the array
            console.log(str.substring(index, length));

            // Recursive Call: call help on right half
            help(Math.ceil((length + index)/2));
        }

        // Base Case: Do Nothing
    }
    help(0);
}
```

Polylogarithmic time

An algorithm is said to run in **polylogarithmic time** if $T(n) = O((\log n)^k)$, for some constant k. For example, matrix chain ordering can be solved in polylogarithmic time on a Parallel Random Access Machine.[4]

Sub-linear time

An algorithm is said to run in **sub-linear time** (often spelled **sublinear time**) if $T(n) = o(n)$. In particular this includes algorithms with the time complexities defined above, as well as others such as the $O(n^{½})$ Grover's search algorithm.

Typical algorithms that are exact and yet run in sub-linear time use parallel processing (as the NC_1 matrix determinant calculation does), non-classical processing (as Grover's search does), or alternatively have guaranteed assumptions on the input structure (as the logarithmic time binary search and many tree maintenance algorithms do). However, formal languages such as the set of all strings that have a 1-bit in the position indicated by the first log(n) bits of the string may depend on every bit of the input and yet be computable in sub-linear time.

The specific term *sublinear time algorithm* is usually reserved to algorithms that are unlike the above in that they are run over classical serial machine models and are not allowed prior assumptions on the input.[5] They are however allowed to be randomized, and indeed must be randomized for all but the most trivial of tasks.

As such an algorithm must provide an answer without reading the entire input, its particulars heavily depend on the access allowed to the input. Usually for an input that is represented as a binary string $b_1,...,b_k$ it is assumed that the algorithm can in time O(1) request and obtain the value of b_i for any i.

Sub-linear time algorithms are typically randomized, and provide only approximate solutions. In fact, the property of a binary string having only zeros (and no ones) can be easily proved not to be decidable by a (non-approximate) sub-linear time algorithm. Sub-linear time algorithms arise naturally in the investigation of property testing.

Linear time

An algorithm is said to take **linear time**, or **O(n)** time, if its time complexity is O(n). Informally, this means that for large enough input sizes the running time increases linearly with the size of the input. For example, a procedure that adds up all elements of a list requires time proportional to the length of the list. This description is slightly inaccurate, since the running time can significantly deviate from a precise proportionality, especially for small values of n.

Linear time is often viewed as a desirable attribute for an algorithm. Much research has been invested into creating algorithms exhibiting (nearly) linear time or better. This research includes both software and hardware methods. In the case of hardware, some algorithms which, mathematically speaking, can never achieve linear time with standard computation models are able to run in linear time. There are several hardware technologies which exploit parallelism to provide this. An example is content-addressable memory. This concept of linear time is used in string matching algorithms such as the Boyer-Moore Algorithm and Ukkonen's Algorithm.

Quasilinear time

An algorithm is said to run in quasilinear time if $T(n)$ = **O(n log^k n)** for any constant k; linearithmic time is the case $k = 1$.[6] Using soft-O notation these algorithms are Õ(n). Quasilinear time algorithms are also o($n^{1+\varepsilon}$) for every $\varepsilon > 0$, and thus run faster than any polynomial in n with exponent strictly greater than 1.

Algorithms which run in quasilinear time, in addition to the linearithmic algorithms listed above, include:

- In-place merge sort, O($n \log^2 n$)
- Quicksort, O($n \log n$), in its randomized version, has a running time that is linearithmic in expectation on the worst-case input. Its non-randomized version has a linearithmic running time only when considering average case complexity.

- Heapsort, O(*n* log *n*), merge sort, introsort, binary tree sort, smoothsort, patience sorting, etc. in the worst case
- Fast Fourier transforms, O(*n* log *n*)
- Monge array calculation, O(*n* log *n*)

Linearithmic time

Linearithmic time is a special case of quasilinear time where the exponent, $k = 1$ on the logarithmic term.

A **linearithmic function** is a function of the form $n \cdot \log n$ (i.e., a product of a linear and a logarithmic term). An algorithm is said to run in **linearithmic time** if $T(n) = \mathbf{O}(n \log n)$.[7] Thus, a linearithmic term grows faster than a linear term but slower than any polynomial in *n* with exponent strictly greater than 1.

In many cases, the $n \cdot \log n$ running time is simply the result of performing a $\Theta(\log n)$ operation *n* times. For example, binary tree sort creates a binary tree by inserting each element of the n-sized array one by one. Since the insert operation on a self-balancing binary search tree takes O(log *n*) time, the entire algorithm takes linearithmic time.

Comparison sorts require at least linearithmic number of comparisons in the worst case because $\log(n!) = \Theta(n \log n)$, by Stirling's approximation. They also frequently arise from the recurrence relation $T(n) = 2\,T(n/2) + O(n)$.

Sub-quadratic time

An algorithm is said to be **subquadratic time** if $T(n) = o(n^2)$.

For example, simple, comparison-based sorting algorithms are quadratic (e.g. insertion sort), but more advanced algorithms can be found that are subquadratic (e.g. Shell sort). No general-purpose sorts run in linear time, but the change from quadratic to sub-quadratic is of great practical importance.

Polynomial time

An algorithm is said to be of **polynomial time** if its running time is upper bounded by a polynomial expression in the size of the input for the algorithm, i.e., $T(n) = O(n^k)$ for some constant *k*.[1][8] Problems for which a deterministic polynomial time algorithm exists belong to the complexity class **P**, which is central in the field of computational complexity theory. Cobham's thesis states that polynomial time is a synonym for "tractable", "feasible", "efficient", or "fast".[9]

Some examples of polynomial time algorithms:

- The quicksort sorting algorithm on *n* integers performs at most An^2 operations for some constant *A*. Thus it runs in time $O(n^2)$ and is a polynomial time algorithm.
- All the basic arithmetic operations (addition, subtraction, multiplication, division, and comparison) can be done in polynomial time.
- Maximum matchings in graphs can be found in polynomial time.

Strongly and weakly polynomial time

In some contexts, especially in optimization, one differentiates between **strongly polynomial time** and **weakly polynomial time** algorithms. These two concepts are only relevant if the inputs to the algorithms consist of integers.

Strongly polynomial time is defined in the arithmetic model of computation. In this model of computation the basic arithmetic operations (addition, subtraction, multiplication, division, and comparison) take a unit time step to perform, regardless of the sizes of the operands. The algorithm runs in strongly polynomial time if [10]

1. the number of operations in the arithmetic model of computation is bounded by a polynomial in the number of integers in the input instance; and
2. the space used by the algorithm is bounded by a polynomial in the size of the input.

Any algorithm with these two properties can be converted to a polynomial time algorithm by replacing the arithmetic operations by suitable algorithms for performing the arithmetic operations on a Turing machine. If the second of the above requirement is not met, then this is not true anymore. Given the integer 2^n (which takes up space proportional to n in the Turing machine model), it is possible to compute 2^{2^n} with n multiplications using repeated squaring. However, the space used to represent 2^{2^n} is proportional to 2^n, and thus exponential rather than polynomial in the space used to represent the input. Hence, it is not possible to carry out this computation in polynomial time on a Turing machine, but it is possible to compute it by polynomially many arithmetic operations.

Conversely, there are algorithms which run in a number of Turing machine steps bounded by a polynomial in the length of binary-encoded input, but do not take a number of arithmetic operations bounded by a polynomial in the number of input numbers. The Euclidean algorithm for computing the greatest common divisor of two integers is one example. Given two integers a and b the running time of the algorithm is bounded by $O((\log\ a + \log\ b)^2)$ Turing machine steps. This is polynomial in the size of a binary representation of a and b as the size of such a representation is roughly $\log\ a + \log\ b$. At the same time, the number of arithmetic operations cannot be bound by the number of integers in the input (which is constant in this case, there are always only two integers in the input). Due to the latter observation, the algorithm does not run in strongly polynomial time. Its real running time depends on the magnitudes of a and b and not only on the number of integers in the input.

An algorithm which runs in polynomial time but which is not strongly polynomial is said to run in **weakly polynomial time**.[11] A well-known example of a problem for which a weakly polynomial-time algorithm is known, but is not known to admit a strongly polynomial-time algorithm, is linear programming. Weakly polynomial-time should not be confused with pseudo-polynomial time.

Complexity classes

The concept of polynomial time leads to several complexity classes in computational complexity theory. Some important classes defined using polynomial time are the following.

- **P**: The complexity class of decision problems that can be solved on a deterministic Turing machine in polynomial time.
- **NP**: The complexity class of decision problems that can be solved on a non-deterministic Turing machine in polynomial time.
- **ZPP**: The complexity class of decision problems that can be solved with zero error on a probabilistic

Turing machine in polynomial time.
- **RP**: The complexity class of decision problems that can be solved with 1-sided error on a probabilistic Turing machine in polynomial time.
- **BPP**: The complexity class of decision problems that can be solved with 2-sided error on a probabilistic Turing machine in polynomial time.
- **BQP**: The complexity class of decision problems that can be solved with 2-sided error on a quantum Turing machine in polynomial time.

P is the smallest time-complexity class on a deterministic machine which is robust in terms of machine model changes. (For example, a change from a single-tape Turing machine to a multi-tape machine can lead to a quadratic speedup, but any algorithm that runs in polynomial time under one model also does so on the other.) Any given abstract machine will have a complexity class corresponding to the problems which can be solved in polynomial time on that machine.

Superpolynomial time

An algorithm is said to take **superpolynomial time** if $T(n)$ is not bounded above by any polynomial. It is $\omega(n^c)$ time for all constants c, where n is the input parameter, typically the number of bits in the input.

For example, an algorithm that runs for 2^n steps on an input of size n requires superpolynomial time (more specifically, exponential time).

An algorithm that uses exponential resources is clearly superpolynomial, but some algorithms are only very weakly superpolynomial. For example, the Adleman–Pomerance–Rumely primality test runs for $n^{O(\log \log n)}$ time on n-bit inputs; this grows faster than any polynomial for large enough n, but the input size must become impractically large before it cannot be dominated by a polynomial with small degree.

An algorithm that requires superpolynomial time lies outside the complexity class **P**. Cobham's thesis posits that these algorithms are impractical, and in many cases they are. Since the P versus NP problem is unresolved, no algorithm for an NP-complete problem is currently known to run in polynomial time.

Quasi-polynomial time

Quasi-polynomial time algorithms are algorithms which run slower than polynomial time, yet not so slow as to be exponential time. The worst case running time of a quasi-polynomial time algorithm is $2^{O((\log n)^c)}$ for some fixed c. The best-known classical algorithm for integer factorization, the general number field sieve, which runs in time about $2^{\tilde{O}(n^{1/3})}$ is *not* quasi-polynomial since the running time cannot be expressed as $2^{O((\log n)^c)}$ for some fixed c. If the constant "c" in the definition of quasi-polynomial time algorithms is equal to 1, we get a polynomial time algorithm, and if it is less than 1, we get a sub-linear time algorithm.

Quasi-polynomial time algorithms typically arise in reductions from an NP-hard problem to another problem. For example, one can take an instance of an NP hard problem, say 3SAT, and convert it to an instance of another problem B, but the size of the instance becomes $2^{O((\log n)^c)}$. In that case, this reduction does not prove that problem B is NP-hard; this reduction only shows that there is no polynomial time algorithm for B unless there is a quasi-polynomial time algorithm for 3SAT (and thus all of NP). Similarly, there are some problems for which we know quasi-polynomial time algorithms, but no polynomial time algorithm is known. Such problems arise in approximation algorithms; a famous example is the directed Steiner tree problem, for which there is a quasi-polynomial time approximation algorithm achieving an approximation factor of $O(\log^3 n)$ (n being the number of vertices), but showing the existence of such a polynomial time algorithm is an open problem.

The complexity class **QP** consists of all problems which have quasi-polynomial time algorithms. It can be defined in terms of DTIME as follows.[12]

$$\mathrm{QP} = \bigcup_{c\in\mathbb{N}} \mathrm{DTIME}(2^{(\log n)^c})$$

Relation to NP-complete problems

In complexity theory, the unsolved P versus NP problem asks if all problems in NP have polynomial-time algorithms. All the best-known algorithms for NP-complete problems like 3SAT etc. take exponential time. Indeed, it is conjectured for many natural NP-complete problems that they do not have sub-exponential time algorithms. Here "sub-exponential time" is taken to mean the second definition presented below. (On the other hand, many graph problems represented in the natural way by adjacency matrices are solvable in subexponential time simply because the size of the input is square of the number of vertices.) This conjecture (for the k-SAT problem) is known as the exponential time hypothesis.[13] Since it is conjectured that NP-complete problems do not have quasi-polynomial time algorithms, some inapproximability results in the field of approximation algorithms make the assumption that NP-complete problems do not have quasi-polynomial time algorithms. For example, see the known inapproximability results for the set cover problem.

Sub-exponential time

The term **sub-exponential time** is used to express that the running time of some algorithm may grow faster than any polynomial but is still significantly smaller than an exponential. In this sense, problems that have sub-exponential time algorithms are somewhat more tractable than those that only have exponential algorithms. The precise definition of "sub-exponential" is not generally agreed upon,[14] and we list the two most widely used ones below.

First definition

A problem is said to be sub-exponential time solvable if it can be solved in running times whose logarithms grow smaller than any given polynomial. More precisely, a problem is in sub-exponential time if for every $\varepsilon > 0$ there exists an algorithm which solves the problem in time $O(2^{n^\varepsilon})$. The set of all such problems is the complexity class **SUBEXP** which can be defined in terms of DTIME as follows.[3][15][16][17]

$$\mathrm{SUBEXP} = \bigcap_{\varepsilon>0} \mathrm{DTIME}\left(2^{n^\varepsilon}\right)$$

Note that this notion of sub-exponential is non-uniform in terms of ε in the sense that ε is not part of the input and each ε may have its own algorithm for the problem.

Second definition

Some authors define sub-exponential time as running times in $2^{o(n)}$.[13][18][19] This definition allows larger running times than the first definition of sub-exponential time. An example of such a sub-exponential time algorithm is the best-known classical algorithm for integer factorization, the general number field sieve, which runs in time about $2^{\tilde{O}(n^{1/3})}$, where the length of the input is *n*. Another example is the best-known algorithm for the graph isomorphism problem, which runs in time $2^{O(\sqrt{(n \log n)})}$.

Note that it makes a difference whether the algorithm is allowed to be sub-exponential in the size of the instance, the number of vertices, or the number of edges. In parameterized complexity, this difference is made explicit by considering pairs (L, k) of decision problems and parameters *k*. **SUBEPT** is the class of all parameterized problems that run in time sub-exponential in *k* and polynomial in the input size *n*:[20]

$$\text{SUBEPT} = \text{DTIME}\left(2^{o(k)} \cdot \text{poly}(n)\right).$$

More precisely, SUBEPT is the class of all parameterized problems (L, k) for which there is a computable function $f : \mathbb{N} \to \mathbb{N}$ with $f \in o(k)$ and an algorithm that decides *L* in time $2^{f(k)} \cdot \text{poly}(n)$.

Exponential time hypothesis

Main article: Exponential time hypothesis

The **exponential time hypothesis** (**ETH**) is that 3SAT, the satisfiability problem of Boolean formulas in conjunctive normal form with at most three literals per clause and with *n* variables, cannot be solved in time $2^{o(n)}$. More precisely, the hypothesis is that there is some absolute constant $c>0$ such that 3SAT cannot be decided in time 2^{cn} by any deterministic Turing machine. With *m* denoting the number of clauses, ETH is equivalent to the hypothesis that *k*SAT cannot be solved in time $2^{o(m)}$ for any integer $k \geq 3$.[21] The exponential time hypothesis implies P ≠ NP.

Exponential time

An algorithm is said to be **exponential time**, if $T(n)$ is upper bounded by $2^{\text{poly}(n)}$, where poly(*n*) is some polynomial in *n*. More formally, an algorithm is exponential time if $T(n)$ is bounded by $O(2^{n^k})$ for some constant *k*. Problems which admit exponential time algorithms on a deterministic Turing machine form the complexity class known as **EXP**.

$$\text{EXP} = \bigcup_{c \in \mathbb{N}} \text{DTIME}\left(2^{n^c}\right)$$

Sometimes, exponential time is used to refer to algorithms that have $T(n) = 2^{O(n)}$, where the exponent is at most a linear function of *n*. This gives rise to the complexity class **E**.

$$\text{E} = \bigcup_{c \in \mathbb{N}} \text{DTIME}\left(2^{cn}\right)$$

Double exponential time

An algorithm is said to be double exponential time if $T(n)$ is upper bounded by $2^{2^{\mathrm{poly}(n)}}$, where poly(n) is some polynomial in n. Such algorithms belong to the complexity class 2-EXPTIME.

$$\text{2-EXPTIME} = \bigcup_{c \in \mathbb{N}} \mathrm{DTIME}(2^{2^{n^c}})$$

Well-known double exponential time algorithms include:

- Decision procedures for Presburger arithmetic
- Computing a Gröbner basis (in the worst case [22])
- Quantifier elimination on real closed fields takes at least double exponential time,[23] and can be done in this time.[24]

See also

- L-notation
- Space complexity

References

1. Sipser, Michael (2006). *Introduction to the Theory of Computation*. Course Technology Inc. ISBN 0-619-21764-2.
2. Mehlhorn, Kurt; Naher, Stefan (1990). "Bounded ordered dictionaries in O(log log N) time and O(n) space". *Information Processing Letters* **35** (4): 183. doi:10.1016/0020-0190(90)90022-P.
3. Babai, László; Fortnow, Lance; Nisan, N.; Wigderson, Avi (1993). "BPP has subexponential time simulations unless EXPTIME has publishable proofs". *Computational Complexity* (Berlin, New York: Springer-Verlag) **3** (4): 307–318. doi:10.1007/BF01275486.
4. Bradford, Phillip G.; Rawlins, Gregory J. E.; Shannon, Gregory E. (1998). "Efficient Matrix Chain Ordering in Polylog Time". *SIAM Journal on Computing* (Philadelphia: Society for Industrial and Applied Mathematics) **27** (2): 466–490. doi:10.1137/S0097539794270698. ISSN 1095-7111.
5. Kumar, Ravi; Rubinfeld, Ronitt (2003). "Sublinear time algorithms" (PDF). *SIGACT News* **34** (4): 57–67. doi:10.1145/954092.954103.
6. Naik, Ashish V.; Regan, Kenneth W.; Sivakumar, D. (1995). "On Quasilinear Time Complexity Theory" (PDF). *Theoretical Computer Science* **148**: 325–349. Retrieved 23 February 2015.
7. Sedgewick, R. and Wayne K (2011). Algorithms, 4th Ed (http://algs4.cs.princeton.edu/home/). p. 186. Pearson Education, Inc.
8. Papadimitriou, Christos H. (1994). *Computational complexity*. Reading, Mass.: Addison-Wesley. ISBN 0-201-53082-1.
9. Cobham, Alan (1965). "The intrinsic computational difficulty of functions". *Proc. Logic, Methodology, and Philosophy of Science II*. North Holland.
10. Grötschel, Martin; László Lovász; Alexander Schrijver (1988). "Complexity, Oracles, and Numerical Computation". *Geometric Algorithms and Combinatorial Optimization*. Springer. ISBN 0-387-13624-X.
11. Schrijver, Alexander (2003). "Preliminaries on algorithms and Complexity". *Combinatorial Optimization: Polyhedra and Efficiency* **1**. Springer. ISBN 3-540-44389-4.
12. *Complexity Zoo*: Class QP: Quasipolynomial-Time (https://complexityzoo.uwaterloo.ca/Complexity_Zoo:Q#qp)
13. Impagliazzo, R.; Paturi, R. (2001). "On the complexity of k-SAT". *Journal of Computer and System Sciences* (Elsevier) **62** (2): 367–375. doi:10.1006/jcss.2000.1727. ISSN 1090-2724.

14. Aaronson, Scott (5 April 2009). "A not-quite-exponential dilemma". *Shtetl-Optimized*. Retrieved 2 December 2009.
15. *Complexity Zoo*: Class SUBEXP: Deterministic Subexponential-Time (https://complexityzoo.uwaterloo.ca/Complexity_Zoo:S#subexp)
16. Moser, P. (2003). "Baire's Categories on Small Complexity Classes". *Lecture Notes in Computer Science* (Berlin, New York: Springer-Verlag): 333–342. ISSN 0302-9743.
17. Miltersen, P.B. (2001). "DERANDOMIZING COMPLEXITY CLASSES". *Handbook of Randomized Computing* (Kluwer Academic Pub): 843.
18. Kuperberg, Greg (2005). "A Subexponential-Time Quantum Algorithm for the Dihedral Hidden Subgroup Problem". *SIAM Journal on Computing* (Philadelphia: Society for Industrial and Applied Mathematics) **35** (1): 188. doi:10.1137/s0097539703436345. ISSN 1095-7111.
19. Oded Regev (2004). "A Subexponential Time Algorithm for the Dihedral Hidden Subgroup Problem with Polynomial Space". arXiv:quant-ph/0406151v1 [quant-ph].
20. Flum, Jörg; Grohe, Martin (2006). *Parameterized Complexity Theory*. Springer. p. 417. ISBN 978-3-540-29952-3. Retrieved 2010-03-05.
21. Impagliazzo, R.; Paturi, R.; Zane, F. (2001). "Which problems have strongly exponential complexity?". *Journal of Computer and System Sciences* **63** (4): 512–530. doi:10.1006/jcss.2001.1774.
22. Mayr,E. & Mayer,A.: The Complexity of the Word Problem for Commutative Semi-groups and Polynomial Ideals. *Adv. in Math.* 46(1982) pp. 305-329
23. J.H. Davenport & J. Heintz: Real Quantifier Elimination is Doubly Exponential. *J. Symbolic Comp.* 5(1988) pp. 29-35.
24. G.E. Collins: Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. Proc. 2nd. GI Conference Automata Theory & Formal Languages (Springer Lecture Notes in Computer Science 33) pp. 134-183

Retrieved from "https://en.wikipedia.org/w/index.php?title=Time_complexity&oldid=685596519"

Categories: Analysis of algorithms | Computational complexity theory | Computational resources

---