

# Lecture 29:
# Tractable and Intractable Problems

Aims:

- To look at the ideas of
  - polynomial and exponential functions and algorithms; and
  - tractable and intractable problems.
- To look at ways of solving intractable problems.



# 29.1. Tractable and Intractable Problems

- Let's start by reminding ourselves of some common functions, ordered by how fast they grow.

| | |
|---|---|
| constant | $O(1)$ |
| logarithmic | $O(\log n)$ |
| linear | $O(n)$ |
| n-log-n | $O(n \times \log n)$ |
| quadratic | $O(n^2)$ |
| cubic | $O(n^3)$ |
| exponential | $O(k^n)$, e.g. $O(2^n)$ |
| factorial | $O(n!)$ |
| super-exponential | e.g. $O(n^n)$ |

- Computer Scientists divide these functions into two classes:

  **Polynomial functions:** Any function that is $O(n^k)$, i.e. bounded from above by $n^k$ for some constant $k$.

  E.g. $O(1)$, $O(\log n)$, $O(n)$, $O(n \times \log n)$, $O(n^2)$, $O(n^3)$

  This is really a different definition of the word 'polynomial' from the one we had in a previous lecture. Previously, we defined 'polynomial' to be any function of the form $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$.

  But here the word 'polynomial' is used to lump together functions that are bounded from above by polynomials. So, $\log n$ and $n \times \log n$, which are not polynomials in our original sense, are polynomials by our alternative definition, because they are bounded from above by, e.g., $n$ and $n^2$ respectively.

  **Exponential functions:** The remaining functions.

  E.g. $O(2^n)$, $O(n!)$, $O(n^n)$



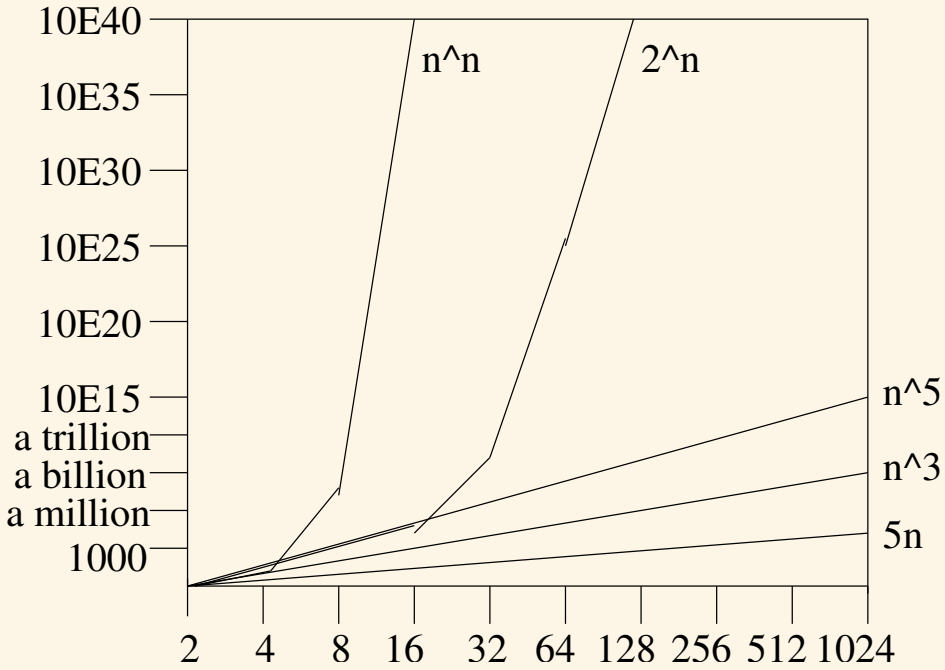
This is a real abuse of terminology. A function of the form $k^n$ is genuinely exponential. But now some functions which are worse than polynomial but not quite exponential, such as $O(n^{\log n})$, are also (incorrectly) called exponential. And some functions which are worse than exponential, such as the super-exponentials, e.g. $O(n^n)$, will also (incorrectly) be called exponential. A better word than 'exponential' would be 'super-polynomial'. But 'exponential' is what everyone uses, so it's what we'll use.

- Why have we lumped functions together into these two broad classes? The next two tables and the graph attempt to show you why.

| | 10 | 50 | 100 | 300 | 1000 |
|---|---|---|---|---|---|
| $5n$ | 50 | 250 | 500 | 1500 | 5000 |
| $n \times \log n$ | 33 | 282 | 665 | 2469 | 9966 |
| $n^2$ | 100 | 2500 | 10000 | 90000 | 1 million (7 digits) |
| $n^3$ | 1000 | 125000 | 1 million (7 digits) | 27 million (8 digits) | 1 billion (10 digits) |
| $2^n$ | 1024 | a 16-digit number | a 31-digit number | a 91-digit number | a 302-digit number |
| $n!$ | 3.6 million (7 digits) | a 65-digit number | a 161-digit number | a 623-digit number | unimaginably large |
| $n^n$ | 10 billion (11 digits) | an 85-digit number | a 201-digit number | a 744-digit number | unimaginably large |

(The number of protons in the known universe has 79 digits.)
(The number of microseconds since the Big Bang has 24 digits.)






(Note that this graph has logarithmic axes.)

- On the basis of this classification of functions into polynomial and exponential, we can classify algorithms:

  **Polynomial-Time Algorithm:** an algorithm whose order-of-magnitude time performance is bounded from above by a polynomial function of $n$, where $n$ is the size of its inputs.



**Exponential Algorithm:** an algorithm whose order-of-magnitude time performance is not bounded from above by a polynomial function of $n$.

- Why do we divide algorithms into these two broad classes? The next table, which assumes that one instruction can be executed every microsecond, attempt to show you why.

| | 10 | 20 | 50 | 100 | 300 |
|---|---|---|---|---|---|
| $n^2$ | $\frac{1}{10000}$ second | $\frac{1}{2500}$ second | $\frac{1}{400}$ second | $\frac{1}{100}$ second | $\frac{9}{100}$ second |
| $n^5$ | $\frac{1}{10}$ second | 3.2 seconds | 5.2 minutes | 2.8 hours | 28.1 days |
| $2^n$ | $\frac{1}{1000}$ second | 1 second | 35.7 years | 400 trillion centuries | a 75-digit number of centuries |
| $n^n$ | 2.8 hours | 3.3 trillion years | a 70-digit number of centuries | a 185-digit number of centuries | a 728-digit number of centuries |

(The Big Bang was approximately 15 billion years ago.)

- And, in a similar way, we can classify problems into two broad classes:

  **Tractable Problem:** a problem that is solvable by a polynomial-time algorithm. The upper bound is polynomial.

  **Intractable Problem:** a problem that cannot be solved by a polynomial-time algorithm. The lower bound is exponential.

- Here are examples of tractable problems (ones with known polynomial-time algorithms):
  - Searching an unordered list



  - Searching an ordered list
  - Sorting a list
  - Multiplication of integers (even though there's a gap)
  - Finding a minimum spanning tree in a graph (even though there's a gap)

- Here are examples of intractable problems (ones that have been proven to have no polynomial-time algorithm).
  - Some of them require a non-polynomial amount of output, so they clearly will take a non-polynomial amount of time, e.g.:
    * Towers of Hanoi: we can prove that any algorithm that solves this problem must have a worst-case running time that is at least $2^n - 1$.
    * List all permutations (all possible orderings) of $n$ numbers.
  - Others have polynomial amounts of output, but still cannot be solved in polynomial time:
    * For an $n \times n$ draughts board with an arrangement of pieces, determine whether there is a winning strategy for White (i.e. a sequence of moves so that, no matter what Black does, White is guaranteed to win). We can prove that any algorithm that solves this problem must have a worst-case running time that is at least $2^n$.

- So you might think that problems can be neatly divided into these two classes. But this ignores 'gaps' between lower and upper bounds. Incredibly, there are problems for which the state of our knowledge is such that the gap spans this coarse division into tractable and intractable. So, in fact, there are three broad classes of problems:
  - Problems with known polynomial-time algorithms.
  - Problems that are provably intractable (proven to have no polynomial-time algorithm).



– Problems with no known polynomial-time algorithm but not yet proven to be intractable.

We'll see some examples of the third category (as well as further examples of the first two categories) in the next lecture.



## 29.2. Solving Intractable Problems

- None of this would matter much if the problems for which we do not have polynomial-time algorithms were theoretical curiosities. Unfortunately, this is not the case. Many real-world problems fall into this category. Unless your inputs are going to be very small, you cannot simply use the known algorithms.
- So what do you do if your problem
  - is provably intractable (proven to have no polynomial-time algorithm), or
  - has no known polynomial-time algorithm even if it is not yet proven intractable?
- Here are the main possibilities:
  - Seek to obtain as much improvement as possible and live hopefully! For example, our backtracking solution to $n$-Queens was probably better than our first solution. Eliminating symmetry in the problem may help further. Incorporating rules-of-thumb ('heuristics') to dynamically decide what to try next may also help. All of these ideas try to make the algorithm work well in practice, on typical instances, while acknowledging that exponential cases are still possible.
  - Solve simpler/restricted versions of the problem. Maybe a solution to a slight variant of the problem would still be useful to you, while possibly avoiding exponential complexity.
  - Use a polynomial-time probabilistic algorithm: one which gives the right answer only with very high probability. So you are giving up on program correctness, in the interests of speed.
  - For optimisation problems, use a polynomial-time approximation algorithm: one which is not guaranteed to find the best answer.



**Acknowledgements:**

The tables and graphs come from [Har92].

Clip Art (of head with bomb) licensed from the Clip Art Gallery on DiscoverySchool.com.



## References

[Har92] D. Harel. *Algorithmics: The Spirit of Computing.* Addison-Wesley, 2nd edition, 1992.