# Computer Science for Biologists

We will be interested in the correctness and efficiency of computer *algorithms*.

We seek algorithms which provably always return the best possible solution to a *well-defined* combinatorial problem.

*Heuristics* are procedures which might return good answers in practice, but are not provably correct.

We seek to extract clean, well-defined problems from the typically messy ``real'' problem to gain insight into it.

This process is analogous to *in vitro* versus *in vivo* experimentation.

## Why do we study algorithms and performance?

Algorithms help us to understand ***scalability***.

Performance often draws the line between what is feasible and what is impossible.

Algorithmic mathematics provides a ***language*** for talking about program behavior.

Performance is the ***currency*** of computing.

The lessons of program performance generalize to other computing resources.

Speed is fun!

## Example: The problem of sorting

***Input:*** sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

***Output:*** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

**Example:**

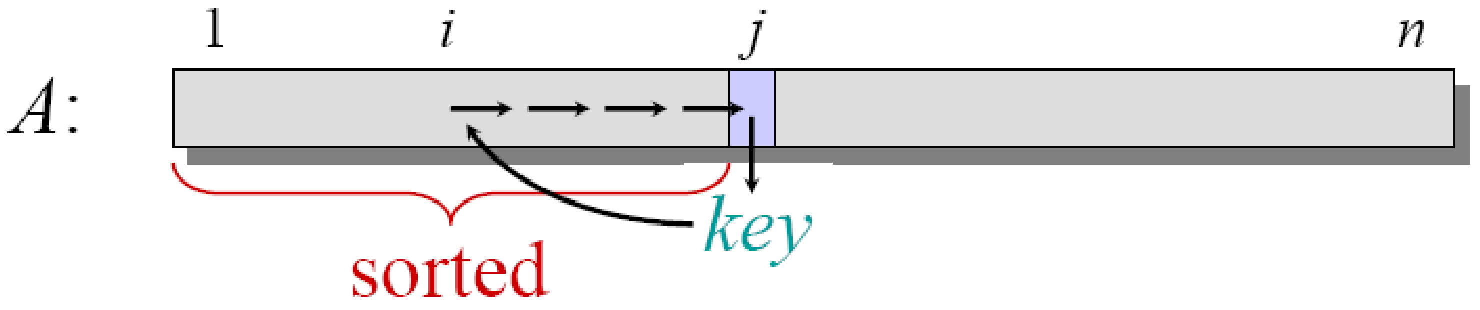
***Input:*** 8 2 4 9 3 6

***Output:*** 2 3 4 6 8 9

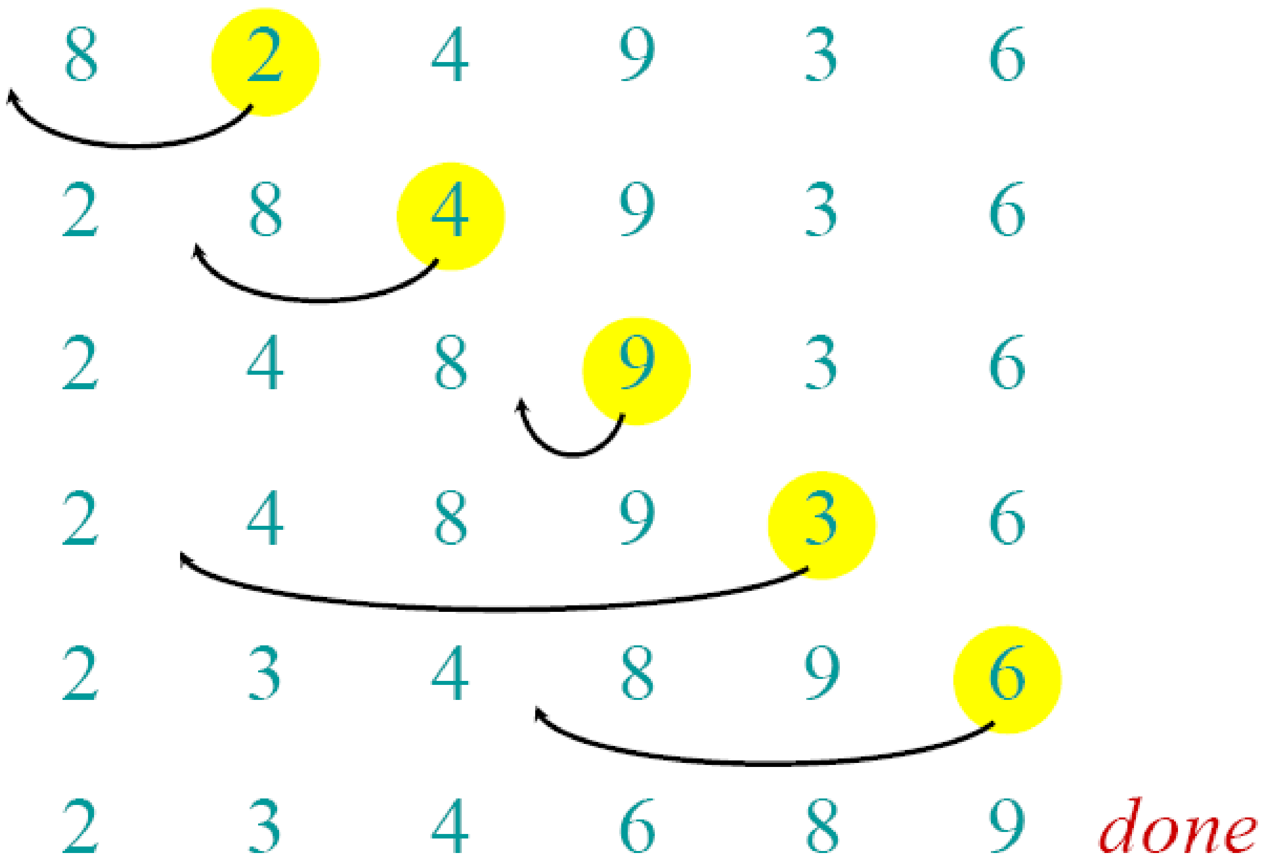
# Insertion sort

“pseudocode”

```
INSERTION-SORT $(A, n)$        $\triangleright A[1 \, . \, . \, n]$
    for $j \leftarrow 2$ to $n$
        do $key \leftarrow A[j]$
           $i \leftarrow j - 1$
           while $i > 0$ and $A[i] > key$
               do $A[i+1] \leftarrow A[i]$
                  $i \leftarrow i - 1$
           $A[i+1] = key$
```



# Insertion sort example



## Running time of insertion sort

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

## Kind of analyses

**Worst-case:** (usually)

- $T(n)$ = maximum time of algorithm on any input of size $n$.

**Average-case:** (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size $n$.
- Need assumption of statistical distribution of inputs.

**Best-case:** (bogus)

- Cheat with a slow algorithm that works fast on *some* input.

## Machine independent time

*What is insertion sort's worst-case time?*

- It depends on the speed of our computer:
    - relative speed (on the same machine),
    - absolute speed (on different machines).

**BIG IDEA:**

- Ignore machine-dependent constants.
- Look at ***growth*** of $T(n)$ as $n \to \infty$ .

**"Asymptotic Analysis"**

## Theta-notation

***Math:***

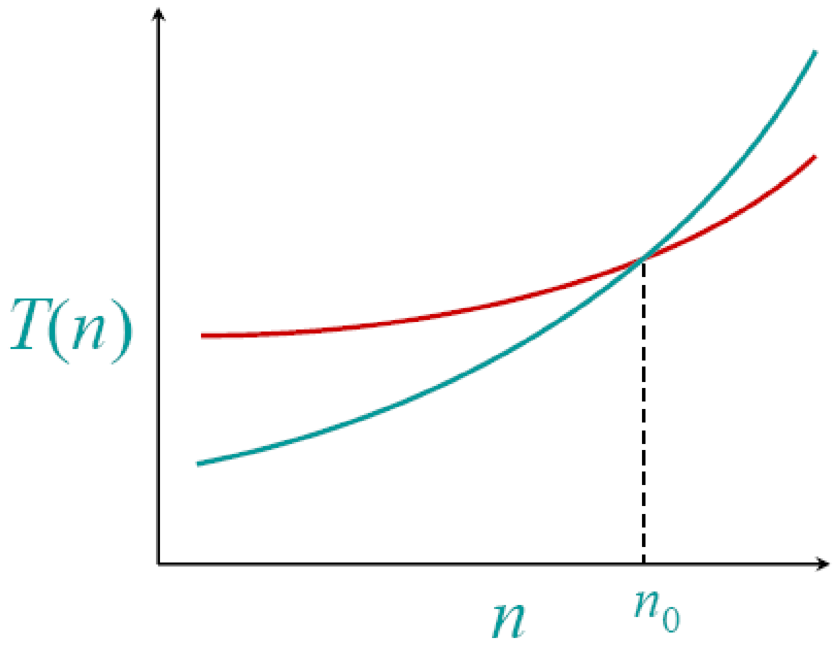
$\Theta(g(n)) = \{ f(n)$ : there exist positive constants $c_1$, $c_2$, and $n_0$ such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$

***Engineering:***

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

## Asymptotic Performance

When $n$ gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm.



- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.

## Insertion sort analysis

***Worst case:*** Input reverse sorted.

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta\left(n^2\right) \qquad \text{[arithmetic series]}$$

***Average case:*** All permutations equally likely.

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta\left(n^2\right)$$

*Is insertion sort a fast sorting algorithm?*

- Moderately so, for small $n$.
- Not at all, for large $n$.

## Asymptotic notation

$O$-notation (upper bounds):

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.

**Example:** $2n^2 = O(n^3)$ $(c = 1, n_0 = 2)$

*functions, not values*

*funny, "one-way" equality*

## Set definition for O-notation

$O(g(n)) = \{ f(n) :$ there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$

**Example:** $2n^2 \in O(n^3)$

## Macro substitution

*Convention:* A set in a formula represents an anonymous function in the set.

**Example:** $f(n) = n^3 + O(n^2)$

means

$f(n) = n^3 + h(n)$

for some $h(n) \in O(n^2)$ .

## Macro substitution

***Convention:*** A set in a formula represents an anonymous function in the set.

**EXAMPLE:** $n^2 + O(n) = O(n^2)$

means

for any $f(n) \in O(n)$:

$n^2 + f(n) = h(n)$

for some $h(n) \in O(n^2)$ .

## Omega-notation

$O$-notation is an *upper-bound* notation. It makes no sense to say $f(n)$ is at least $O(n^2)$.

$\Omega(g(n)) = \{ f(n)$ : there exist constants $c > 0$, $n_0 > 0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0 \}$

**EXAMPLE:** $\sqrt{n} = \Omega(\lg n)$ $(c = 1,\ n_0 = 16)$

## Theta-notation

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

**Example:** $\frac{1}{2}n^2 - 2n = \Theta(n^2)$

## o-notation and ω-notation

$O$-notation and $\Omega$-notation are like $\leq$ and $\geq$.
$o$-notation and $\omega$-notation are like $<$ and $>$.

$o(g(n)) = \{ f(n) :$ for any constant $c > 0$,
there is a constant $n_0 > 0$
such that $0 \leq f(n) < cg(n)$
for all $n \geq n_0$ $\}$

**EXAMPLE:** $2n^2 = o(n^3)$ $(n_0 = 2/c)$

## o-notation and ω-notation

$O$-notation and $\Omega$-notation are like $\leq$ and $\geq$.
$o$-notation and $\omega$-notation are like $<$ and $>$.

$\omega(g(n)) = \{ f(n) :$ for any constant $c > 0$,
there is a constant $n_0 > 0$
such that $0 \leq cg(n) < f(n)$
for all $n \geq n_0$ $\}$

**Example:** $\sqrt{n} = \omega(\lg n)$ $\quad (n_0 = 1 + 1/c)$

# Exact string matching

**Input**: A text string $T$, where $|T| = n$, and a pattern string $P$, where $|P| = m$.

**Output**: An index $i$ such that $T_{i+j-1} = P_j$ for all $1 \leq j \leq m$, i.e. showing that $P$ is a substring of $T$.

The following brute force search algorithm always uses at most $n \times m$ steps:

```
for $i = 1$ to $n - m + 1$ do
        $j = 1$
        while $(T[i + j - 1] == P[j])$ and $(j \leq m))$
                do $j = j + 1$
        if $(j > m)$ print "pattern at position ", $i$
```

## Exact string matching

This algorithm might use only $n$ steps if we are lucky, e.g. $T$=*aaaaaaaaaaa*, and $P$=*bbbbbbb*.

We might need ~ $n \times m$ steps if we are unlucky, e.g. $T$=*aaaaaaaaaaa*, and $P$=*aaaaab*.

We can't say what happens ``in practice'', so we settle for a worst case analysis.

By being more clever, we can reduce the worst case running time to $O(n+m)$.

Certain generalizations won't change this, like stopping after the first occurrence of the pattern.

Certain other generalizations seem more complicated, like matching with gaps.

## Algorithm Complexity

We use the Big oh notation to state an upper bound on the number of steps that an algorithm takes in the worst case.

Thus the brute force string matching algorithm is O(mn), or takes *quadratic* time.

A *linear* time algorithm, i.e. O(n+m) , is fast enough for almost any application.

A quadratic time algorithm is usually fast enough for small problems, but not big ones, since $1{,}000^2 = 1{,}000{,}000$ steps is reasonable but $1{,}000{,}000^2$ is not.

An exponential-time algorithm, i.e. $O(2^n)$ or O(n!), can only be fast enough for tiny problems, since $2^{20}$ and 10! are already up to 1,000,000.

``A billion here, a billion there, and soon you are talking about real money'' - Senator Everett Dirksen

## NP-Completeness

Unfortunately, for many problems, there is no known *polynomial* algorithm.

Even worse, most of these problems can be proven *NP-complete*, meaning that no such algorithm can exist!

At the 1999 RECOMB conference, there was a rebellion by biologists tired of seeing all their problems shown NP-complete.

But proving a problem NP-complete can be a useful thing to do, because it focuses our attention on heuristics and tells us why it is difficult.

NP-completeness proofs work by showing that the target problem is as ``hard'' as some famous hard problem, e.g. satisfiability, vertex cover, Hamiltonian cycle.

## NP-Completeness – Whimsical example

Suppose that you are employed in the halls of industry.

Your boss assigns you a problem for designing a method determining whether or not any given set of specifications for a new component can be met and, if so, for constructing a design that meets them.

You are the company’s chief algorithm designer and you are in charge of finding an efficient algorithm for doing this.

After defining the problem exactly, you pull down your reference books and plunge into the task with great enthusiasm.

Weeks later, your office filled with mountains of crumpled-up scratch paper, your enthusiasm has lessened considerably and you could not come up with any algorithm substantially better than searching through all possible designs.

You do not want to go back to your boss office and report:

# NP-Completeness – Whimsical example

"I can't find an efficient algorithm, I guess I'm just too dumb."

## NP-Completeness – Whimsical example

To avoid serious damage to your position within the company, it would be much better if you could prove that the problem is inherently intractable, that no algorithm could possibly solve it quickly.

You then could stride confidently into the boss's office and proclaim:

# NP-Completeness – Whimsical example

“I can't find an efficient algorithm, because no such algorithm is possible!”

## NP-Completeness – Whimsical example

Unfortunately, proving inherent intractability can be just as hard as finding efficient algorithms.

Even the best theoreticians have been stymied in their attempts to obtain such proofs for commonly encountered hard problems.

However you have discovered something almost as good. The theory of NP-completeness provides many straightforward techniques for proving that a given problem is "just as hard" as a large number of other problems that are widely recognized as being difficult and that have been confounding the experts for years.

Armed with these techniques, you might be able to prove that your problem is NP-complete and, hence, that it is equivalent to all these other hard problems. Then you could march into your boss's office and announce:

# NP-Completeness – Whimsical example

“I can't find an efficient algorithm, but neither can all these famous people.”

## NP-Completeness – Whimsical example

At the very least, this would inform your boss that it would do no good to fire you and hire another expert on algorithms!

Discovering that a problem is NP-complete is usually just the beginning of work on that problem. The need for solving a problem won't disappear overnight simply because their problem is known to be NP-complete.

The knowledge that it is NP-complete does provide valuable information about what lines of approach have the potential of being most productive.

Certainly the search for an efficient, exact algorithm should be accorded low priority. It is now more appropriate to concentrate on other, less ambitious, approaches.

## NP-Complete problems – Approaches

You might look for efficient algorithms that solve various special cases of the general problem.

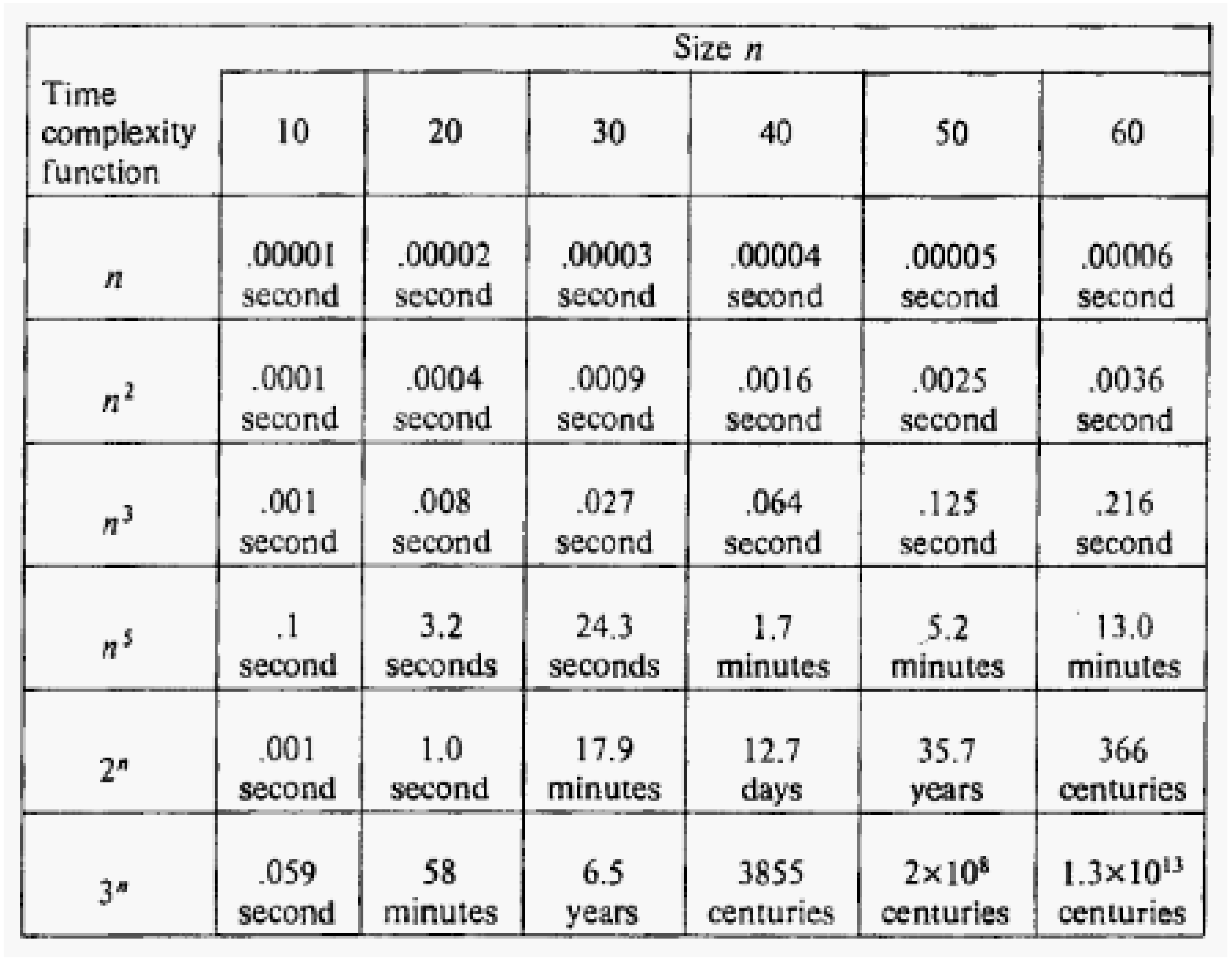
You might look for algorithms that, though not guaranteed to run quickly, seem likely to do so most of the time.

You might relax the problem, looking for a  fast algorithm that merely finds designs that meet most of the component specifications.

You might want to find a “good enough” solution, instead of the best solution possible.

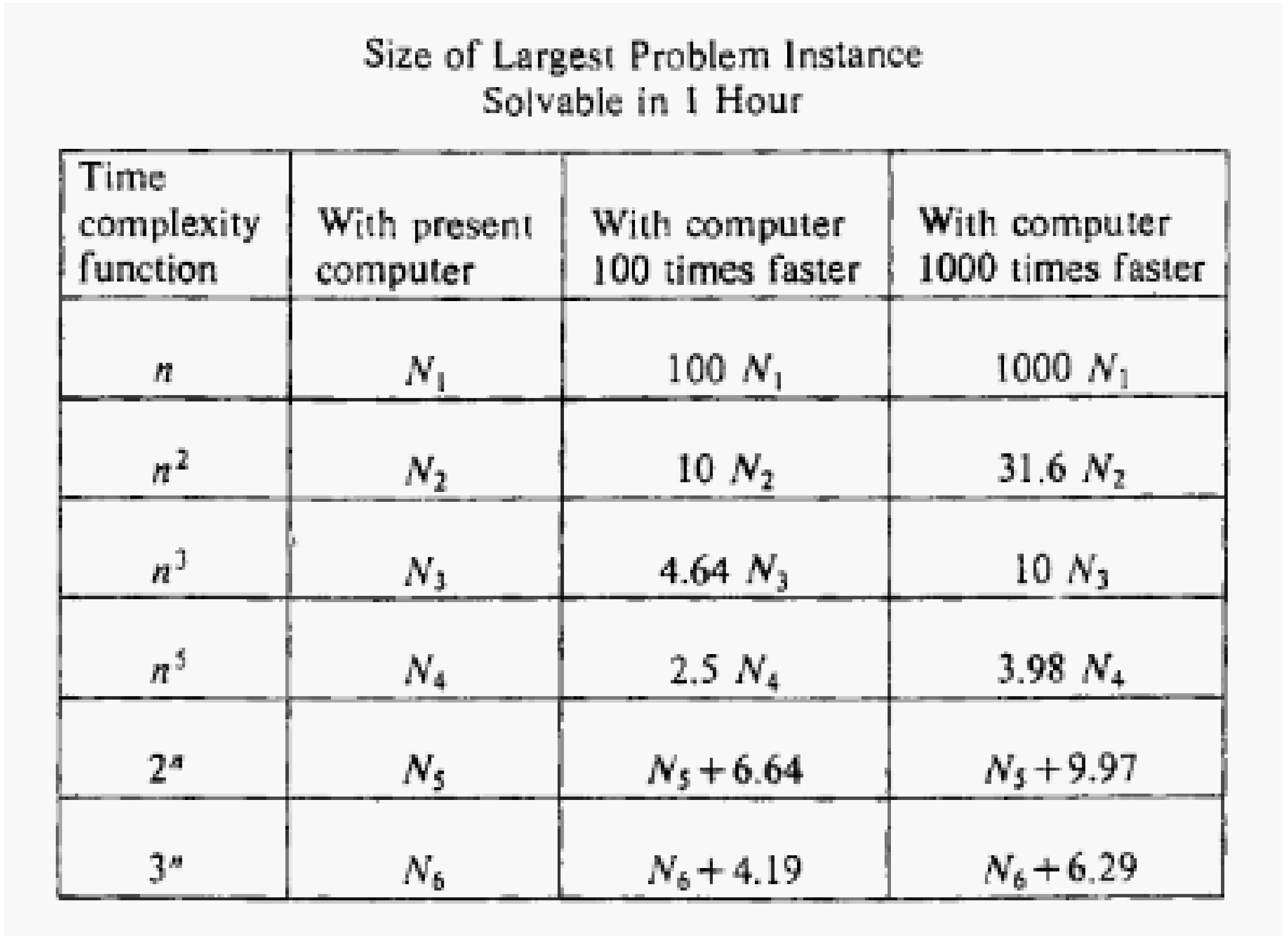
In short, the primary application of the theory of NP-completeness is to assist algorithm designers in directing their problem-solving efforts toward those approaches that have the greatest likelihood of leading to useful algorithms.

# Time complexity



| Time complexity function | Size $n$ | | | | | |
|---|---|---|---|---|---|---|
| | 10 | 20 | 30 | 40 | 50 | 60 |
| $n$ | .00001 second | .00002 second | .00003 second | .00004 second | .00005 second | .00006 second |
| $n^2$ | .0001 second | .0004 second | .0009 second | .0016 second | .0025 second | .0036 second |
| $n^3$ | .001 second | .008 second | .027 second | .064 second | .125 second | .216 second |
| $n^5$ | .1 second | 3.2 seconds | 24.3 seconds | 1.7 minutes | 5.2 minutes | 13.0 minutes |
| $2^n$ | .001 second | 1.0 second | 17.9 minutes | 12.7 days | 35.7 years | 366 centuries |
| $3^n$ | .059 second | 58 minutes | 6.5 years | 3855 centuries | $2\times10^8$ centuries | $1.3\times10^{13}$ centuries |

# Time complexity



Size of Largest Problem Instance
Solvable in 1 Hour

| Time complexity function | With present computer | With computer 100 times faster | With computer 1000 times faster |
|---|---|---|---|
| $n$ | $N_1$ | $100\ N_1$ | $1000\ N_1$ |
| $n^2$ | $N_2$ | $10\ N_2$ | $31.6\ N_2$ |
| $n^3$ | $N_3$ | $4.64\ N_3$ | $10\ N_3$ |
| $n^5$ | $N_4$ | $2.5\ N_4$ | $3.98\ N_4$ |
| $2^n$ | $N_5$ | $N_5 + 6.64$ | $N_5 + 9.97$ |
| $3^n$ | $N_6$ | $N_6 + 4.19$ | $N_6 + 6.29$ |