TOOLS FOR ALGORITHM ANALYSIS

- **Overview**
  - Measuring Complexity
  - Best-Case, Worst-Case, and Average-Case Complexity
  - Asymptotic Analysis
  - Amortized Analysis
- **References**

  - Kenneth A. Berman, Jerome L. Paul, "Algorithms: Sequential, Parallel, and Distributed", Thomsen Course Technology (2005)
  - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms", 4th Edition, The MIT Press (2022)
  - Russ Miller, Laurence Boxer, "Algorithms Sequential & Parallel - A Unified Approach", 2nd Edition, Charles River Media, Inc. (2005)
  - Scott Meyers: Effective Modern C++. O'Reilly (2014)
  - Anthony Williams: C++ Concurrency in Action - Practical Multithreading. Manning Publications Co. (2012)

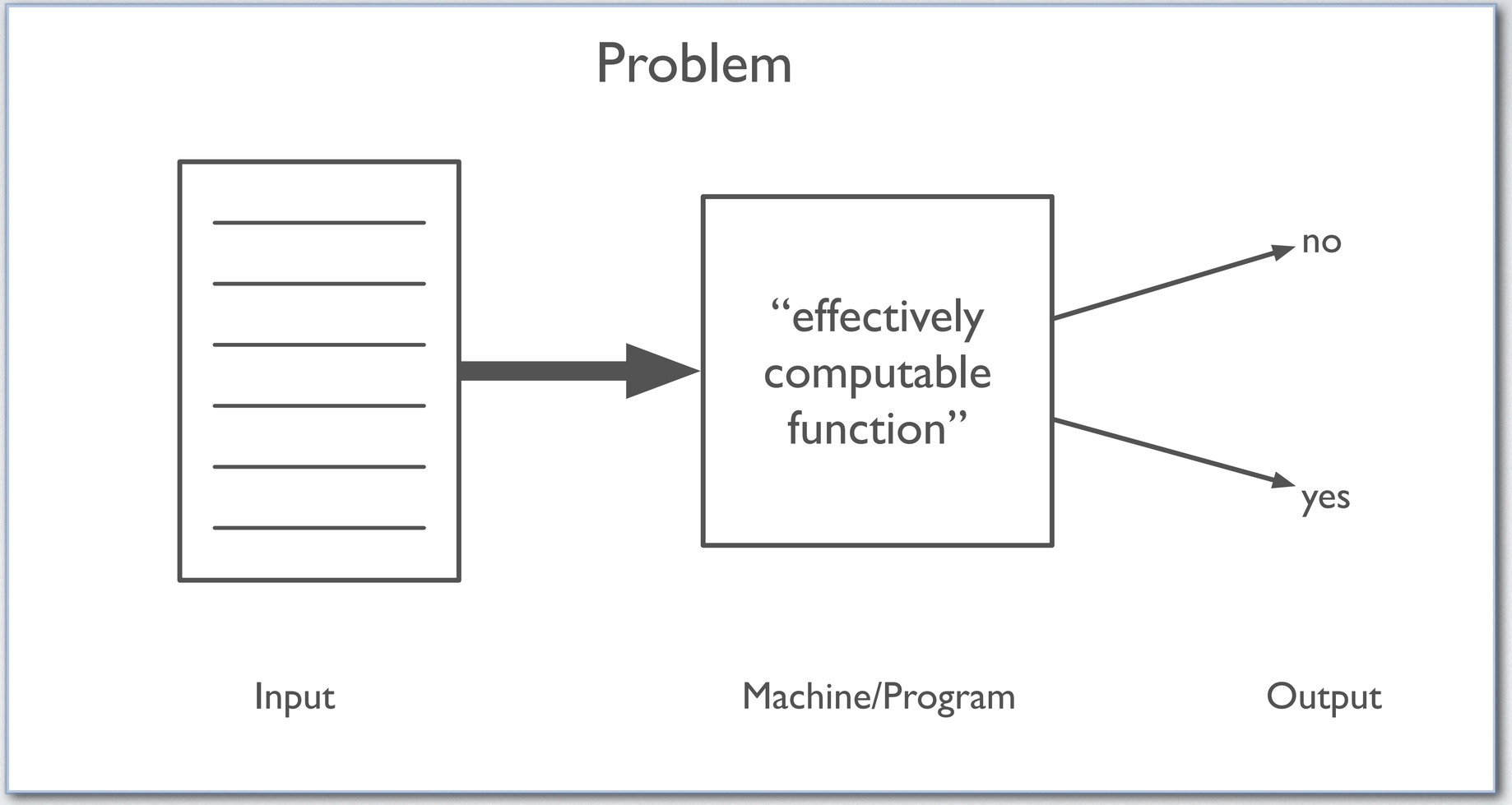
WHAT IS COMPUTABLE?

- Computation is usually modeled as a mapping from inputs to outputs, carried out by a "formal machine" or program, which processes its input in a sequence of steps.

- An "effectively computable" function can be computed in a finite amount of time using finite resources.

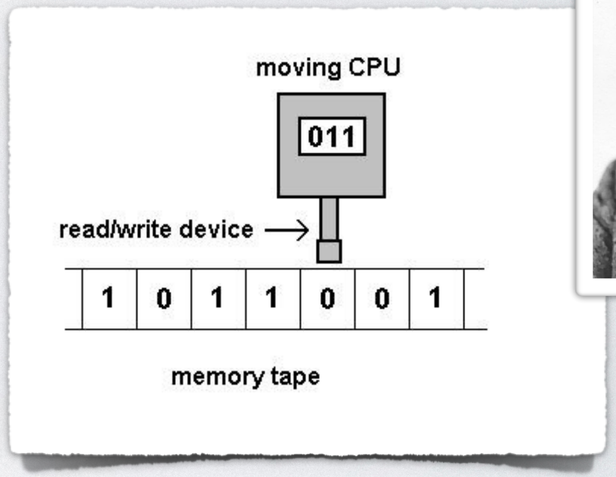
- There are many so-called intractable problems for which no efficient algorithm exists. Intractable problems are common. Most intractable problems have an algorithm that provides a solution: brute-force search. However, this algorithm does not provide an efficient solution and is not feasible for computation with anything more than the smallest input (e.g., traveling salesman, bin packing).

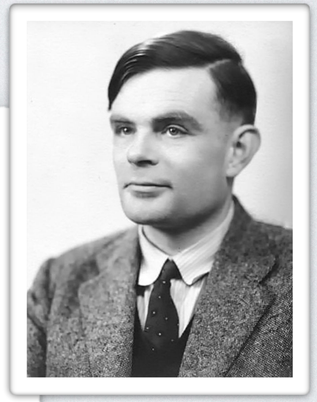
ABSTRACT MACHINE MODEL



WHAT CAN BE COMPUTED: TURING MACHINE

- A Turing machine is an abstract representation of a computing device. It consists of a read/write head that scans a (possibly infinite) one-dimensional (bi-directional) tape divided into squares, each of which is inscribed with a 0 or 1 (possibly more symbols).
- Computation begins with the machine, in a given "state", scanning a square. It erases what it finds there, prints a 0 or 1, moves to an adjacent square, and goes into a new state until it moves into HALT.
- This behavior is completely determined by three parameters:
  - the state the machine is in,
  - the number on the square it is scanning, and
  - a table of instructions.





DEFINITION OF A TURING MACHINE

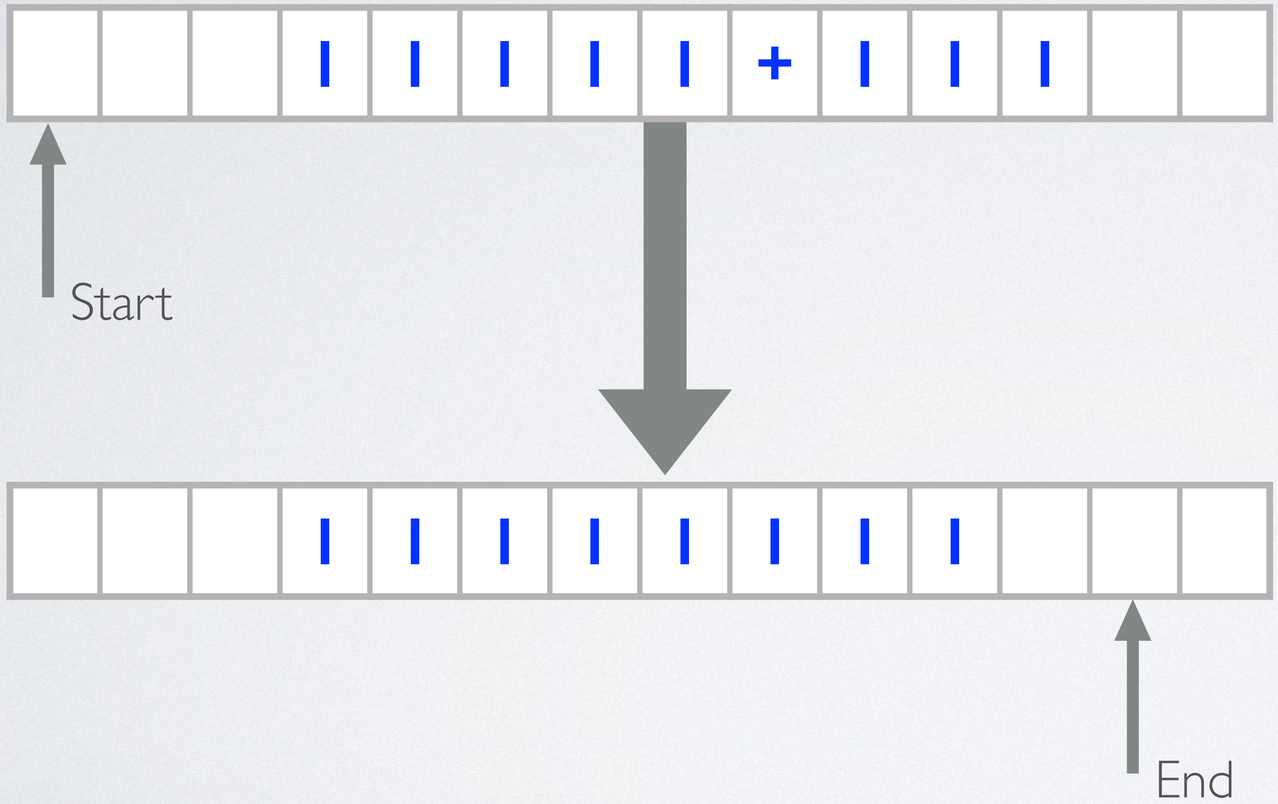
- A Turing machine is a septuple $(Q, \Gamma, \gamma, \Sigma, q0, F, \sigma)$ with:
  - a set $Q = \{q0, q1, \ldots\}$ of states,
  - a set Γ is a finite, non-empty set of tape symbols,
  - a blank symbol $\gamma \in \Gamma$ (the only symbol to occur infinitely often)
  - a set $\Sigma \subseteq \Gamma \setminus \{ \gamma \}$ of input symbols (to appear as initial tape contents),
  - a designated start state $q0$.
  - a subset $F \subseteq Q$ called the accepting states,
  - a partial function $\sigma: (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times \{ R, L \}$ called the transition function.
- A Turing machine halts if it enters a state in F or if σ is undefined for the current state and tape symbol.

ADDITION WITH A TURING MACHINE

- Σ = {“I”,“+”,“ ”} - the tape symbols

Q\Γ	“ ”	“I”	“+”
1	1/“ ”/R	2/“I”/R	
2		2/“I”/R	3/“+”/R
3		4/“+”/L	
4			5/“I”/R
5	6/“ ”/L	4/“+”/L	5/“+”/R
6			HALT/“ ”/R

5+3



Q\Γ	“ ”	“I”	“+”
1	1/“ ”/R	2/“I”/R	
2		2/“I”/R	3/“+”/R
3		4/“+”/L	
4			5/“I”/R
5	6/“ ”/L	4/“+”/L	5/“+”/R
6			HALT/“ ”/R

CHURCH'S THESIS

- **Church's Thesis: It is impossible to build a computing device that is more powerful than a Turing machine.**
- Church's thesis cannot be proven because "effectively computable" is an intuitive notion, not a mathematical one.
- It can only be refuted by given a counter-example – a machine that can solve a problem not computable by a Turing machine.
- So far, all models of effectively computable functions are equivalent to Turing machines.

HALTING PROBLEM

- A problem that cannot be solved by a Turing machine in finite time (or any equivalent formalism) is called uncomputable.
  - Assuming Church's thesis is true, an uncomputable problem cannot be solved by any real computer.
- **The Halting Problem**:
  - Given an arbitrary Turing machine and its input tape, will the machine eventually halt?
  - The Halting Problem is provably uncomputable – which means that it cannot be solved in practice.

THE ACKERMANN FUNCTION

- The Ackermann function is the simplest example of a well-defined total function that is computable but not primitive recursive. That is, the Ackermann function cannot be computed by a computer program whose loops are all "for" loops (i.e, it is impossible to determine an upper bound of the total number of iterations for every loop before entering the loop).

- The function f(x) = A(x, x), while Turing computable, grows much faster than polynomials or exponentials. The definition is:

$$
\begin{aligned}
A(0, n) &= n+1 \\
A(m+1, 0) &= A(m, 1) \\
A(m+1, n+1) &= A(m, A(m+1, n))
\end{aligned}
$$

EXAMPLE VALUES OF THE ACKERMANN FUNCTION

A(m, n)	n = 0	n = 1	n = 2	n = 3	n = 4	n = 5
m = 0	1	2	3	4	5	6
m = 1	2	3	4	5	6	7
m = 2	3	5	7	9	11	13
m = 3	5	13	29	61	125	253
m = 4	13	65533	$2^{65536}-3$	A(3, A(4, 2))	A(3, A(4, 3))	A(3, A(4, 4))
m = 5	65533	A(4, A(5, 0))	A(4, A(5, 1))	A(4, A(5, 2))	A(4, A(5, 3))	A(4, A(5, 4))
m = 6	A(5, 65533)	A(5, A(6, 0))	A(5, A(6, 1))	A(5, A(6, 2))	A(5, A(6, 3))	A(5, A(6, 4))

A(4,2) - NUMBER WITH 19,729 DECIMAL DIGITS

A(4,2) =
20035299304068464649790723515602557504478254755697514192650169737108940595563114530895061308809333481010382343429072631818229493821188126688695063647615470291650418719163515879663472194429309279820843091048559905701593189596395248633723672030029169695921561087649488892540908059114570376752085002066715637023661263597471448071117748158809141357427209671901518362825606180914588526998261414250301233911082736038437678764490432059603791244909057075603140350761625624760318637931264847037437829549756137709816046144133086921181024859591523801953310302921628001605686701056516467505680387415294638422448452925373614425336143737290883037946012747249584148649159306472520151556939226281806916507963810641322753072671439981585088112926289011342377827055674210800700652839633221550778312142885516755540733451072131124273995629827197691500548839052238043570458481979563931578535100189920000241419637068135598404640394721940160695176901561197269823378900017
...
133663377137843441619405312144529185518013657555866761501937302969193207612000925506508158327550849934076879725236998702356793102680413674571895664143185267905471716996299036301554564509004480278905570196832831363071899769915316667920895876857229060091547291963638167359667395997571032601557192023734858052112811745861006515259888384311451189488055212914577569914657753004138471712457796504817585639507289533753975582208777750607233944558789590571915673 3

COMPARING ALGORITHMS

THE "BEST" ALGORITHM

- There are usually multiple algorithms to solve any particular problem.
- The notion of the "best" algorithm may depend on many different criteria:
  - Structure, composition, and readability,
  - Time required to implement,
  - Extensibility,
  - Space requirements, and
  - Running time requirements.

FIRST EXAMPLE

Algorithm A runs 2 minutes, and algorithm B takes 1 minute and 45 seconds to complete for the same input.

Is B "better" than A? ➠ Not necessarily!:

- We have tested A and B only on one (fixed) input set. Another input set might result in a different runtime behavior.
- Algorithm A might have been interrupted by another process.
- Algorithm B might have been run on a different computer.

A reasonable time and space approximation should be machine-independent.

MEASURING COMPLEXITY (TIME)

- To analyze the complexity of an algorithm, we can identify a basic operation and count how many times the algorithm performs that operation. This approach is not dependent on a particular computer.

- We measure the complexity of an algorithm as a function of the input size.

- Some care is required when input size is taken for measurement. Consider the problem of computing b^k for $k > 0$. A naive algorithm takes $k - 1$ multiplications, so we might take $n = k$ as this input size. However, $n = k$ is a single number, not the size of the aggregate. The power b^k can have hundreds of digits. In such case, $k - 1$ multiplications would not complete in the allotted time frame ($k - 1 \geq 2^{127} - 1$ multiplications may require hundreds of years). The actual number of multiplications required is approximately equal to 2^n.

RUNNING TIME IN TERMS OF INPUT SIZE

- What is one of the most interesting aspects about algorithms?
  - How many seconds does it take to complete for a particular input size n?
  - How does an increase in input size affect running time?
- An algorithm requires
  - Constant time if the running time remains the same as the size n changes,
  - Linear time if the running time increases proportionally to size n.
  - Exponential time if the running time increases exponentially for size n.

GENERAL PRINCIPLES

- We use T(n) to represent the running time of an algorithm operating on a data set of size n.
- Ignore machine-dependent constants: We do not concern ourselves with how fast an individual processor executes a machine instruction.
- Look at growth of T(n) as $n \rightarrow \infty$: Even an inefficient algorithm will often finish its work in an acceptable time when operating on a small data set. We are usually interested in T(n), the running time of an algorithm for large n.
- Growth Rate: Asymptotic analysis implies that we are interested in the general behavior a function f(n) exhibits as the input parameter n gets large. Lower order terms and constant factors become irrelevant in terms of considering the growth rate of f(n) as n gets large and are, hence, ignored.

A SIMPLE EXAMPLE: $\sum_{i=1}^{n} i^3$

```
1: size_t sum( size_t n ) noexcept
2: {
3:     size_t Result = 0;
4:
5:     for ( size_t i = 1; i <= n; i++ )
6:         Result += i * i * i;
7:
8:     return Result;
9: }
```

➠ **T(n) = 6n + 4**; linear running time, O(n)

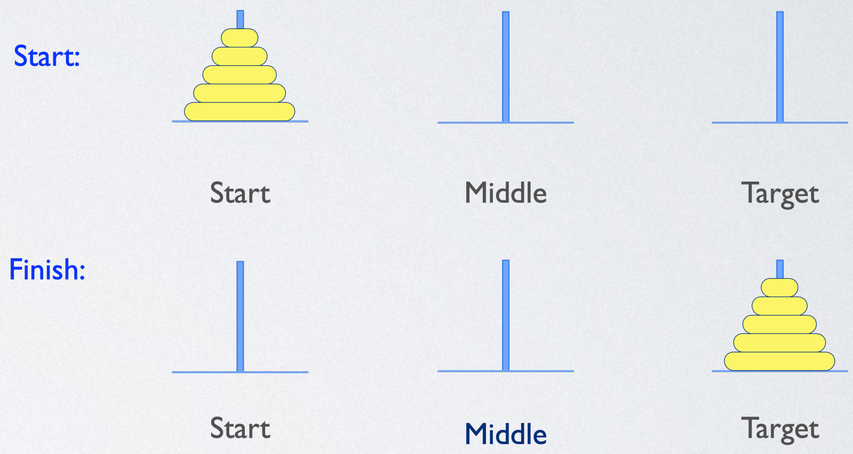
- We count the basic operations in each line:
  - line 1: Zero, the declaration requires no operations.
  - line 3 & 8: One operation each.
  - line 5: Hidden costs for initializing i, testing i <= n, and incrementing i; one operation initialization, n+1 operations for all tests, and n operations for all increments: 2n + 2.
  - line 6: Four operations, two multiplications, one addition, and one assignment, that run n times: 4n.

CONSTANT RUNNING TIME SOLUTION

$$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2} \right)^2$$

TOWERS OF HANOI: RECURSIVE PROCEDURE

```
procedure TON( n, S, T, M )
begin
  if n > 0 then
    TON( n-1, S, M, T )
    d(T) ← d(S)
    TON( n-1, M, T, S )
end
```



TOWERS OF HANOI COMPLEXITY

- The body of TON requires $T(n) = 2T(n-1) + 1$ operations for an input size n.
- We have
  - $T(n-1) = 2T(n-2) + 1 = 2*(2T(n-3) + 1) + 1 = 2^2T(n-3) + 2^1 + 1$
  - $T(n-2) = 2T(n-3) + 1 = 2*(2T(n-4) + 1) + 1$
  - …
  - $T(0) = 1$
- Solving the recursive equations, we obtain
  - $T(n) = 2^n + 2^{(n-1)} + \ldots 2^1 + 1 = 2^{(n+1)} - 1$; exponential running time, $O(2^n)$

GENERAL COUNTING RULES

FOR-LOOP

```
for ( initializer; condition; expression )
    statement
```

- The running time for a for-loop is at most the running time of the statement inside the for-loop times the number of iterations.
- Let C be the running time of the statement. Then a for-loop has a running time of, at most, Cn or O(n).
- Single for-loops have a linear running time.

NESTED FOR-LOOPS

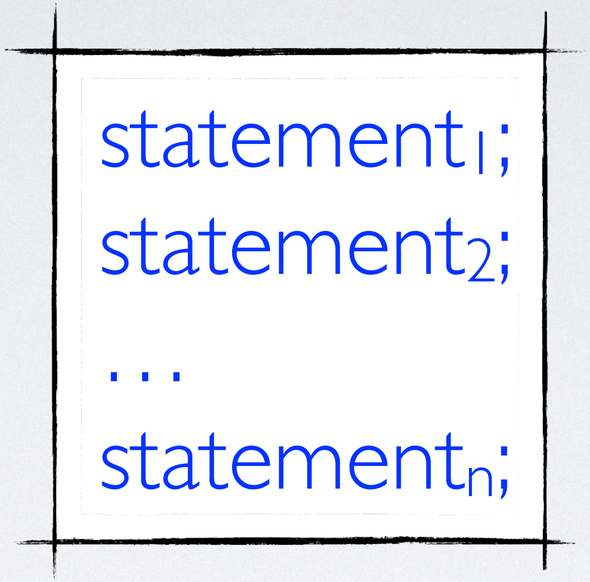
```
for ( initializer1; condition1; expression1 )
  for ( initializer2; condition2; expression2 )
    statement
```

- The running time for nested for-loops is, at most, the running time of the statement multiplied by the product of the sizes of all the for-loops.

- Let C be the running time of the statement. Then, a nested for-loop has a running time of at most Cn*n or $O(n^2)$ (the asymptotic upper bound for a nested for-loop has quadratic running time).

- Nested for-loops have quadratic running time. Any additional nesting level adds one factor. A k-nested loop requires polynomial time or $O(n^k)$.

STATEMENT SEQUENCE



- The running time for statement sequence is the sum of each statement.
- Let t_i be the running time for each statement. Then a statement sequence has a running time of at most $t_1 + t_2 + ... t_n$ or $O(m)$ where $m = max(t_1, t_2, ..., t_n)$. That is, the running time of a statement sequence is taken to be that which "dominates" the running time.

BRANCHING

```
if ( condition )
   true-statement;
else
   false-statement;
```

- The running time for an if-then-else statement is, at most, the running time of the condition plus the larger running times of the true and false statement.

- This can yield an overestimate, but it never produces an underestimate.

INPUTS AND NUMBER OF BASIC OPERATIONS

- An input I to an algorithm is the data, such as numbers, character strings, and records, on which the algorithm is run.
- The set $\mathcal{F}_n$ denotes the set of all inputs of size n to an algorithm.
- For $I \in \mathcal{F}_n$, $\tau(I)$ represents the number of basic operations performed when the algorithm is executed with input I.

THE EFFECT OF INPUTS

- For some algorithms, the number of basic operations performed is the same for any input of size n.
- For most algorithms, the number of basic operations performed can differ for two inputs of the same size.
- We consider three scenarios: best-case, worst-case, and average complexity. In some scenarios, the worst-case complexity of an algorithm is the most relevant, for example, when deadlines must be met.

BEST-CASE COMPLEXITY

- The best-case complexity of an algorithm is the function B(n) such that B(n) equals the minimum value of τ(▮), where ▮ varies over all inputs of size n:

$$B(n) = \min \{ \tau(▮) \mid ▮ \in \mathcal{F}_n \}$$

- B(n) is typically easy to compute but not particularly useful. It is only employed to capture the lower bound of the average complexity of an algorithm.

WORST-CASE COMPLEXITY

- The worst-case complexity of an algorithm is the function W(n) such that W(n) equals the maximum value of τ(▮), where ▮ varies over all inputs of size n:

$$W(n) = \max \{ \tau(▮) \mid ▮ \in \mathcal{F}_n \}$$

- W(n) is very useful when estimating the running time T(n) of an algorithm. It provides the upper bound for the complexity of an algorithm and, hence, a means for algorithm selection.

AVERAGE COMPLEXITY

- The average complexity of an algorithm with a finite input set $\mathcal{F}_n$ is defined as:

$$A(n) = \sum_{I \in \mathcal{F}_n} \tau(I) p(I) = E[\tau]$$

where $p(I)$ is the probability of $I \in \mathcal{F}_n$ occurring as the input to the algorithm. If every input is equally likely, then $p(I) = 1/|\mathcal{F}_n|$. E[τ] is the expected number of basic operations.

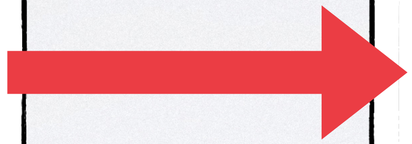
- A(n) is rarely used directly. A more practical variant is

$$A(n) = \sum_{i=1}^{W(n)} i p_i = E[\tau]$$

- where p_i denotes the probability of the algorithm performing exactly i basic operations.

BUBBLE SORT

```
1.   t ← N - 1
2.   while t > 0
3.     BOUND ← t
4.     t ← 0
5.     while 0 ≤ j < BOUND
6.       if A[j] > A[j+1]
7.         A[j] :=: A[j+1]
8.         t ← j
9.       j ← j + 1
```



```
template<typename T, typename C = std::greater<T>>
void doBubbleSort( T aArray[], size_t aSize, C aIsOutOfOrder = C{} )
{
   size_t t = aSize - 1;                                   // C1

   while ( t > 0 )                                         // C2
   {
      size_t BOUND = t;                                    // C3

      t = 0;                                               // C4

      for ( size_t j = 0; j < BOUND; j++ )                 // C5/C9
      {
         if ( aIsOutOfOrder( aArray[j], aArray[j+1] ) )    // C6
         {
            std::swap( aArray[j], aArray[j+1] );           // C7
            t = j;                                         // C8
         }
      }
   }
}
```

- Donald E. Knuth: The Art of Computer Programming - Volume 3 / Sorting and Searching. 2nd Edition. Addition-Wesley. 1998, pp. 107-109.

BUBBLE SORT ANALYSIS

```
1.  t ← N - 1
2.  while t > 0
3.    BOUND ← t
4.    t ← 0
5.    while 0 ≤ j < BOUND
6.      if A[j] > A[j+1]
7.        A[j] :=: A[j+1]
8.        t ← j
9.      j ← j + 1
```

BOUND	Tests
n - 1	**n**
n - 2	**n - 1**
…	**…**
2	**3**
1	**2**
0	exit condition

Cost	Times
c_1	1
c_2	n
c_3	$n-1$
c_4	$n-1$
c_5	$\sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1$
c_6	$\sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2}$
c_7	$\sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2}$
c_8	$\sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2}$
c_9	$\sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2}$

$$T_{\text{Bubble Sort}}(N) = c_1 + c_2 n + (c_3 + c_4)(n-1) + c_5(\sum_{i=2}^{n} i) + (c_6 + c_9)(\sum_{i=2}^{n} (i-1)) + (c_7 + c_8)(\sum_{i=2}^{n} (i-1))$$

BUBBLE SORT: WORST-CASE RUNNING TIME

$$T_{\text{Bubble Sort}}(N) = c_1 + c_2 n + (c_3 + c_4)(n-1) + c_5(\sum_{i=2}^{n} i) + (c_6 + c_9)(\sum_{i=2}^{n}(i-1)) + (c_7 + c_8)(\sum_{i=2}^{n}(i-1))$$

- The worst case occurs if the array is in reverse sorted order:

$$T^{W}_{\text{Bubble Sort}}(N) = c_1 + c_2 n + (c_3 + c_4)(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + (c_6 + c_7 + c_8 + c_9)\left(\frac{n(n-1)}{2}\right)$$

- The terms n(n+1) and n(n-1) are quadratic and dominate the running time.
- Bubble Sort, in the worst case, is $O(n^2)$, the asymptotic upper bound has quadratic running time.

BUBBLE SORT: BEST-CASE RUNNING TIME

$$T_{\text{Bubble Sort}}(N) = c_1 + c_2 n + (c_3 + c_4)(n-1) + c_5(\sum_{i=2}^{n} i) + (c_6 + c_9)(\sum_{i=2}^{n}(i-1)) + (c_7 + c_8)(\sum_{i=2}^{n}(i-1))$$

- The best case occurs if the array is already sorted:
  - Lines 7 and 8 are never run. Hence $c_7 = c_8 = 0$.
  - The variables t and BOUND initially are equal to n-1. After the first execution of the while-loop starting at line 5, both t and BOUND are equal to 0. Hence, the outer loop exits.

$$T^{B}_{\text{Bubble Sort}}(N) = c_1 + c_2 n + (c_3 + c_4)(n-1) + c_5 n + (c_6 + c_9)(n-1)$$

- The term n is linear and dominates the running time.
- Bubble Sort, in the best case, is Ω(n), the asymptotic lower bound has linear running time.
- The best case is due to an optimization proposed by Donald E. Knuth.

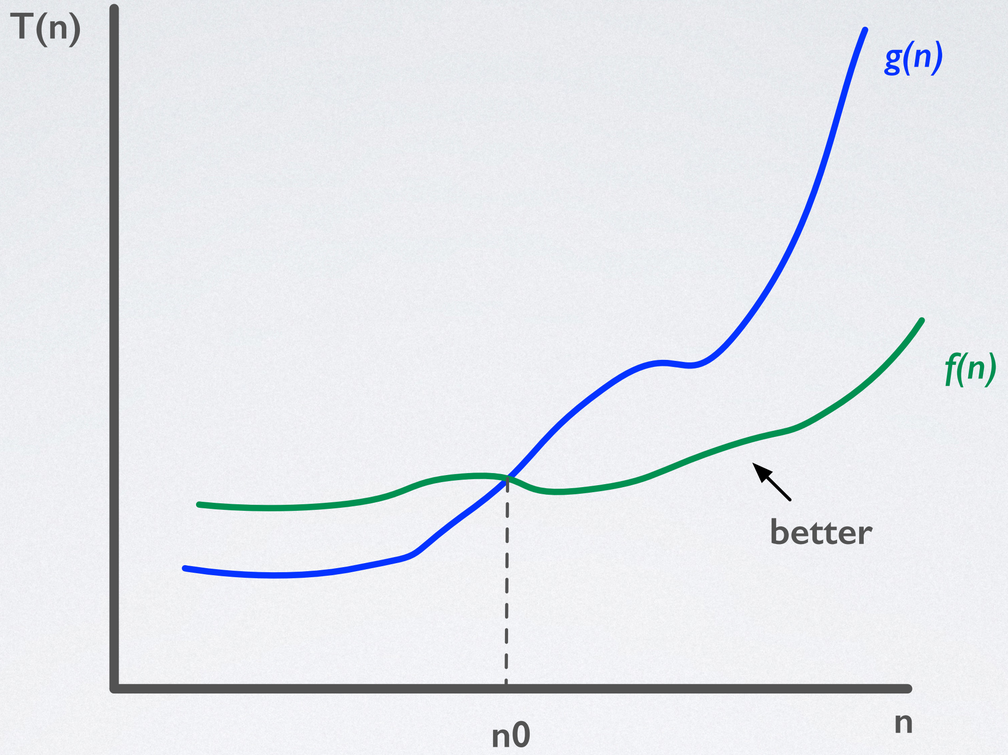
BUBBLE SORT: RUNNING TIME

- Bubble Sort is $O(n^2)$ and $\Omega(n)$.
- However, Bubble Sort is very slow, in general. Moreover, it performs redundant comparisons, and elements can never move more than one position per pass.
- Donald E. Knuth concludes: "… bubble sort seems to have nothing to recommend it, expect a catchy name …".
- Many modern textbooks don't even discuss Bubble Sort anymore.
- Insertion Sort, even though it is also $O(n^2)$ and $\Omega(n)$, does a better job.

ALGORITHM COMPARISON

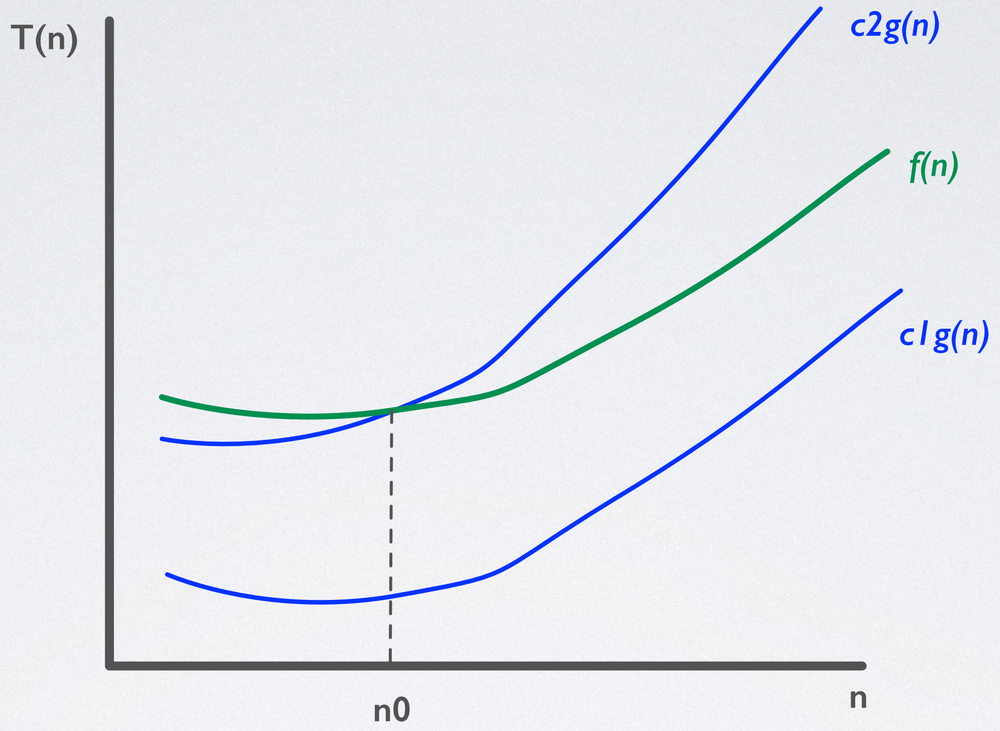
Algorithm/Problem	B(n)	A(n)	W(n)
Towers of Hanoi	2^n	2^n	2^n
Linear Search	1	n	n
Binary Search	1	log n	log n
Min, Max, MinMax	n	n	n
Insertion Sort	n	n^2	n^2
Merge Sort	n log n	n log n	n log n
Heap Sort	n log n	n log n	n log n
Quick Sort	n log n	n log n	n^2

GROWTH RATE OF F(N) AND G(N)



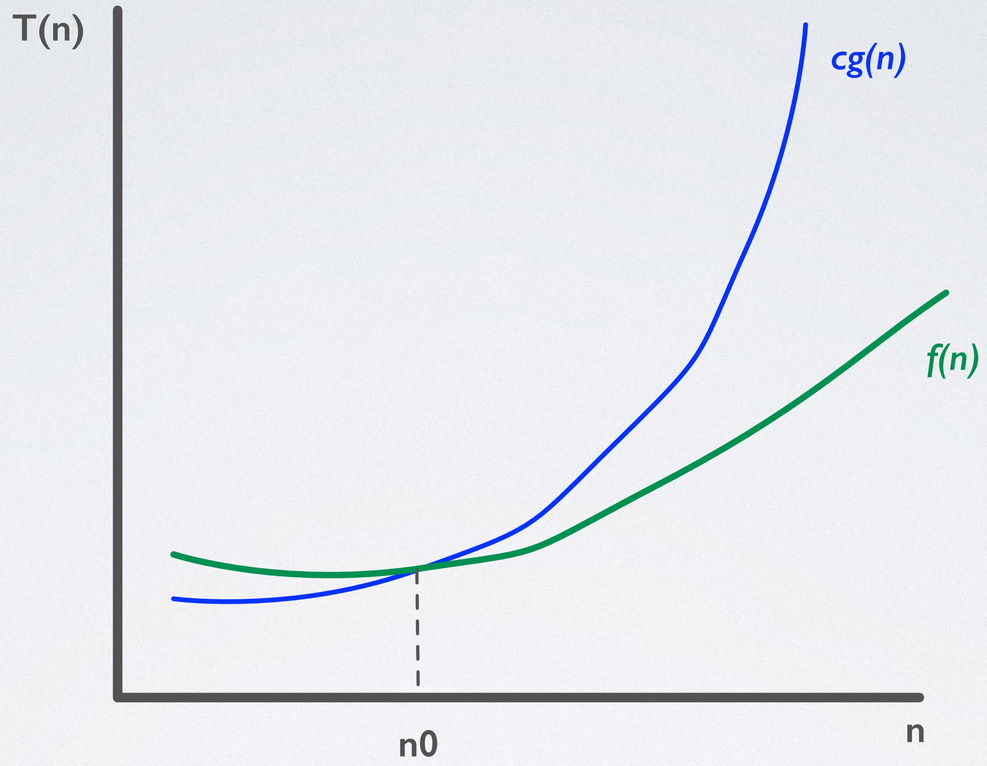
- For large values of n, an algorithm with a running time of $T(n) = f(n)$ is typically more desirable than an algorithm with a running time of $T(n) = g(n)$. Large means $n \geq n_0$.

$F(N) = \Theta(G(N))$



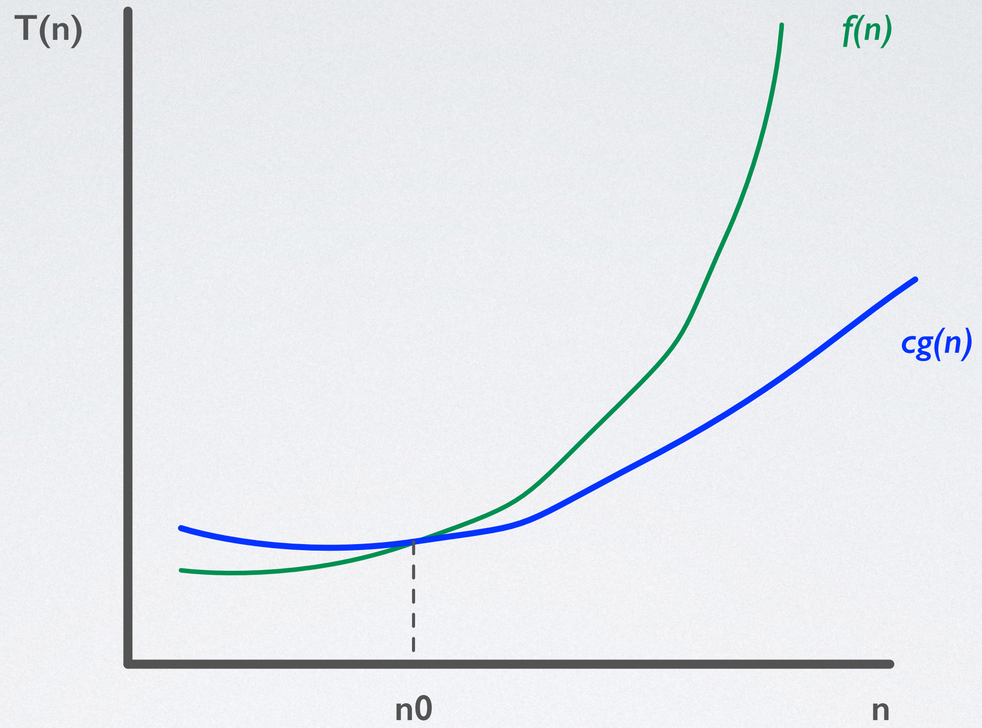
- $f(n) = \Theta(g(n))$ (read "f of n is Theta of g of n") if and only if there exists positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ whenever $n \geq n_0$.

F(N) = O(G(N))



- f(n) = O(g(n)) (read "f of n is big oh of g of n") if and only if there exists positive constants c and n_0 such that f(n) ≤ cg(n) whenever n ≥ n_0.

$F(N) = \Omega(G(N))$



- $f(n) = \Omega(g(n))$ (read "f of n is big omega of g of n") if and only if there exists positive constants c and n_0 such that $cg(n) \leq f(n)$ whenever $n \geq n_0$.

CONVENTIONS

- Θ, O, and Ω are set-valued functions. Therefore, it would be appropriate to write $(3n^2 + 2) \in \Theta(n^2)$.

- However, it is common to write $(3n^2 + 2) = \Theta(n^2)$, which is not correct in the mathematical sense. It reads as

  "3 n squared plus 2 is theta of n squared."

- The set-valued functions Θ, O, and Ω are referred to as asymptotic notation. Therefore, we want the simplest and best representative of an Θ, O, and Ω function.

ASYMPTOTIC RUNNING TIME ESTIMATION

- Asymptotic notation ignores lower-order terms:
  - Lower order terms in the computation steps count functions concerned with initialization, secondary arguments, etc.
- Asymptotic does not consider the multiplier in higher-order terms:
  - These terms are machine-dependent.

CONSTANT RUNNING TIME

- Algorithm A requires 2,000,000 steps: O(1)

- As a young boy, the later mathematician Carl Friedrich Gauss was asked by his teacher to add up the first hundred numbers to keep him quiet for a while. As we know today, this did not work out, since:

$$sum(n) = n(n+1)/2$$

which is O(1).

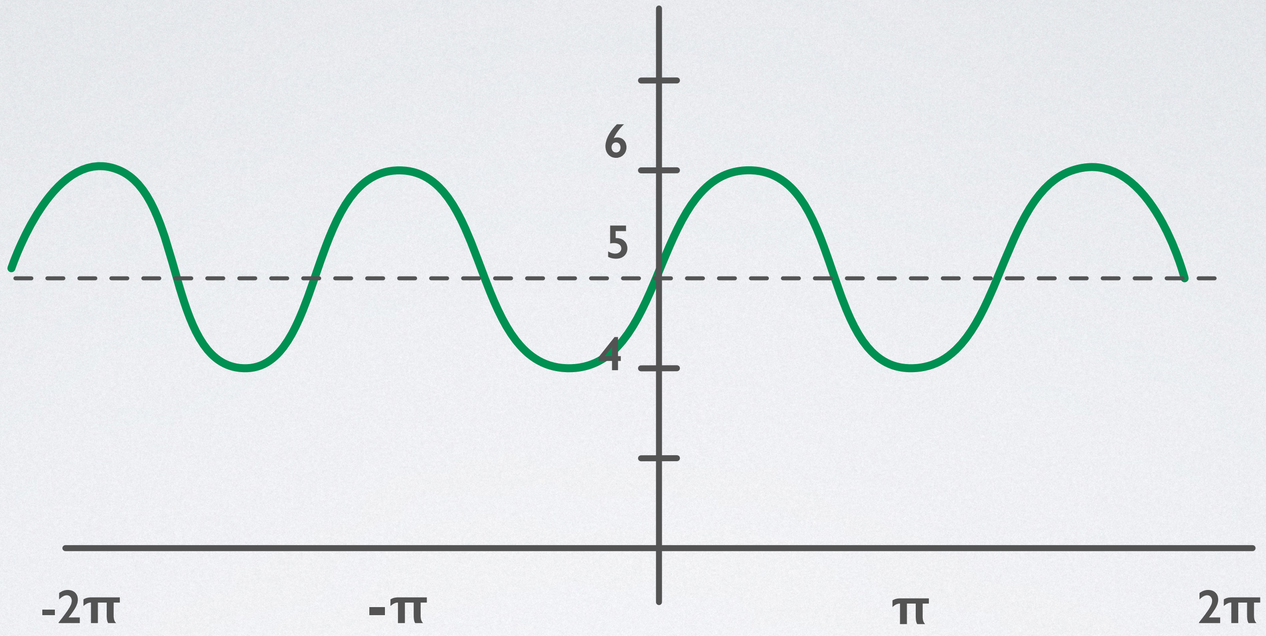
POLYNOMIAL RUNNING TIME

- $4n^2 + 3n + 1 = O(n^2)$
  - Ignore lower-order terms: $4n^2$
  - Ignore constant coefficients: n^2
- $a^k n^k + a^{k-1} n^{k-1} + \ldots + a^1 n^1 + a^0 n^0 = O(n^k)$
  - Ignore a^k and $a^{k-1} n^{k-1} + \ldots + a^1 n^1 + a^0 n^0$: n^k

LOGARITHMIC AND EXPONENTIAL RUNNING TIME

- $\log_{10} n = \log_2 n / \log_2 10 = O(\log n)$
  - Ignore base: $\log n$
- $345n^{4536} + n(\log n) + 2^n = O(2^n)$
  - Ignore lower-order terms $345n^{4536}$ and $n(\log n)$: 2^n

F(T) = 5 + SIN(T)



- Given f(t) = 5 + sin(t) and g(t) = 1, then 5 + sin(t) = Θ(1) because 4 ≤ 5 + sin(t) ≤ 6.
- Also, f(t) = O(1) and f(t) = Ω(1), but the best choice for notation is to write f(t) = Θ(1) because Θ conveys more information than either O or Ω.

SOME ASYMPTOTIC RELATIONSHIPS

- $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$
- $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$
- $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

ASYMPTOTIC ANALYSIS AND LIMIT

- To determine the relationship between functions f and g, it is often useful to examine

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = L$$

- The possible outcomes and implications are:
  - L = 0: g(n) grows faster than f(n).
  - L = ∞: f(n) grows faster than g(n).
  - L ≠ 0 is finite: f(n) and g(n) grow at the same rate.
  - L does not exist: this technique cannot be used to compare f(n) and g(n).

HIERARCHY OF ORDERS

- **Fastest**: $O(1)$ - constant-size lookup table
  $O(\log n)$ - binary search
  $O(n)$ - min, max
  $O(n \log n)$ - merge sort
  $O(n^2)$ - insertion sort
  $O(n^2 \log n)$ - naive construction of a suffix array
  $O(n^3)$ - standard polygon triangulation
  $O(2^n)$ - recursive Fibonacci sequence
- **Slowest**: $O(n!)$ - traveling salesman