# Algorithmic Patterns

## Overview

- Algorithm Efficiency
- Solution Strategies

## References


- Bruno R. Preiss: Data Structures and Algorithms with Object-Oriented Design Patterns in C++. John Wiley & Sons, Inc. (1999)
- Russ Miller and Laurence Boxer: Algorithms Sequential & Parallel - A Unified Approach. 2nd Edition. Charles River Media (2005)
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: Introduction to Algorithms. 3rd Edition. The MIT Press (2009)

# 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
  - Time requirements

# Time Analysis

- Example:

  Algorithm A runs 2 minutes and algorithm B takes 1 minutes and 45 second 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.

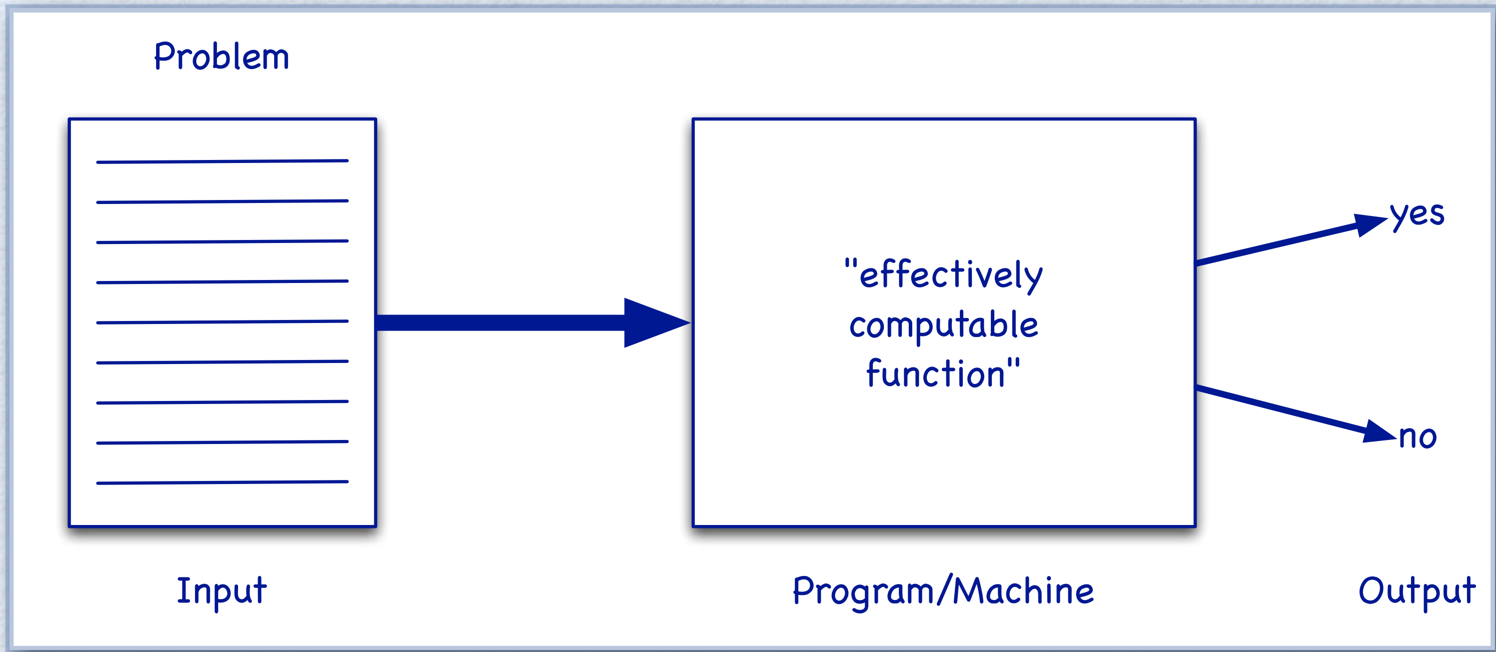
# Running Time in Terms of Input Size

- What is the one of most interesting aspect about algorithms?
  - How many seconds does it take to complete for a particular input size n?
  - How does an increase of size n effect the 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 with respect to size n.

# What is computable?

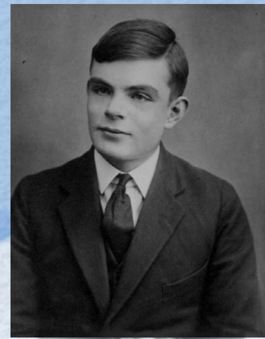
- Computation is usually modeled as a mapping from inputs to outputs, carried out by a "formal machine", or program, which processes its input in a sequence of steps.
- An "effectively computable" function is one that can be computed in a finite amount of time using finite resources.

# Abstract Machine Model

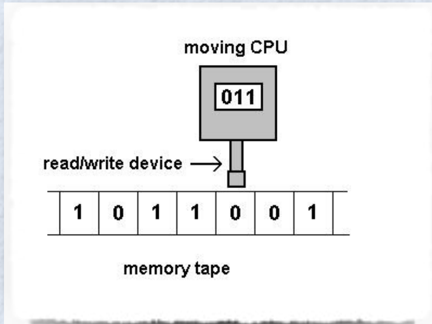



- Church's Thesis: It is not possible to build a machine that is more powerful than a Turing machine.

# 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.



# Formal Definition of a Turing Machine

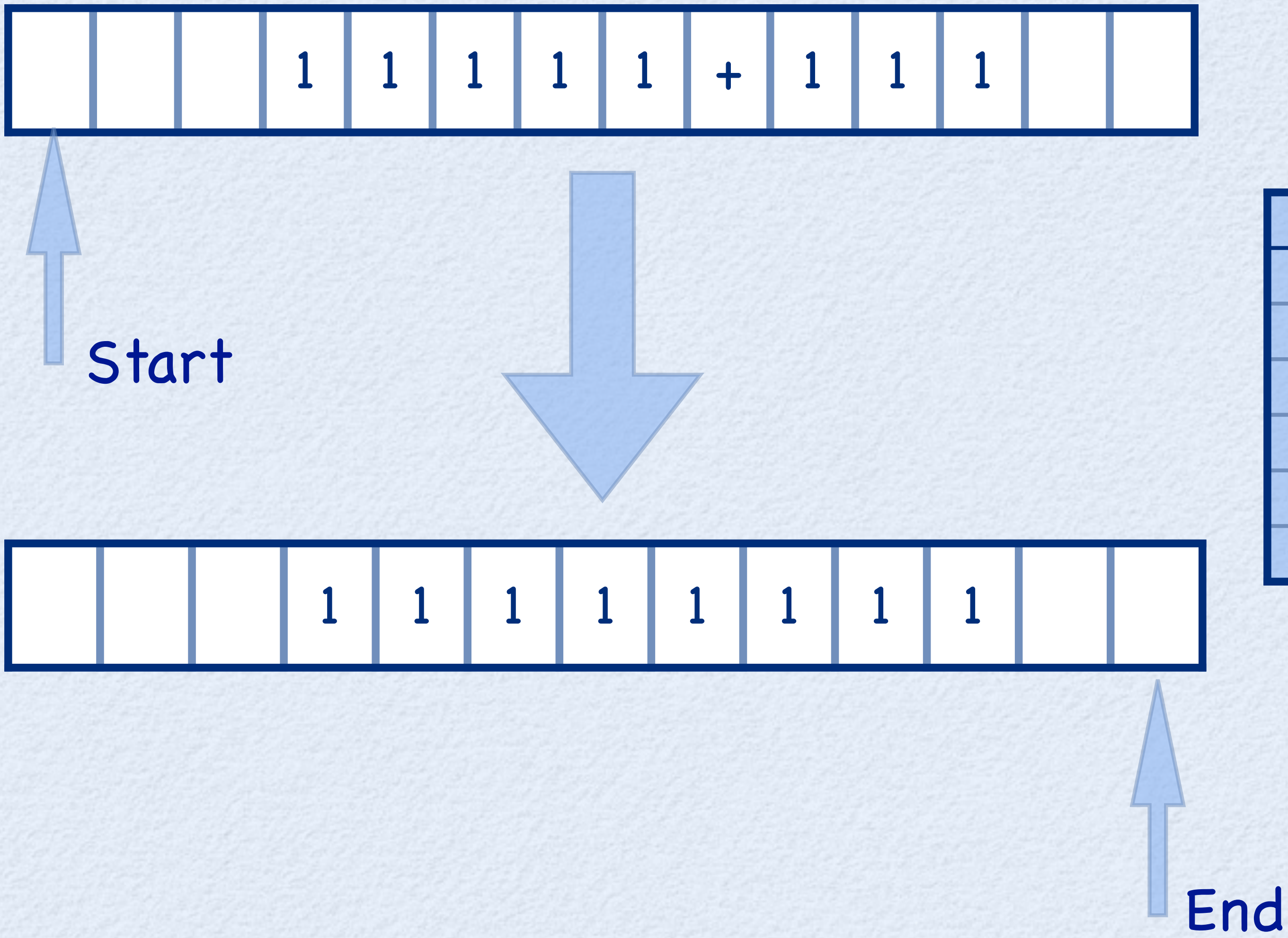
- A Turing machine is a septuple $(Q, \Gamma, \gamma, \Sigma, q_0, F, \sigma)$ with:
  - a set $Q = \{q_0, q_1, \ldots\}$ of states,
  - a set $\Gamma$ is a finite, non-empty set of tape symbols,
  - a blank symbol $\gamma \in \Gamma$ (the only symbol to occur infinitely often)
  - a set $\Sigma \subseteq \Gamma \setminus \{ \gamma \}$ of input symbols (to appear as initial tape contents),
  - a designated start state $q_0$.
  - a subset $F \subseteq Q$ called the accepting states,
  - a partial function $\sigma : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{ R, L \}$ called the transition function.
- A Turing machine halts if it enters a state in F or if $\sigma$ is undefined for the current state and tape symbol.

# Addition on a Turing Machine

- $\Sigma$ = {"1", "+", " "} - the tape symbols

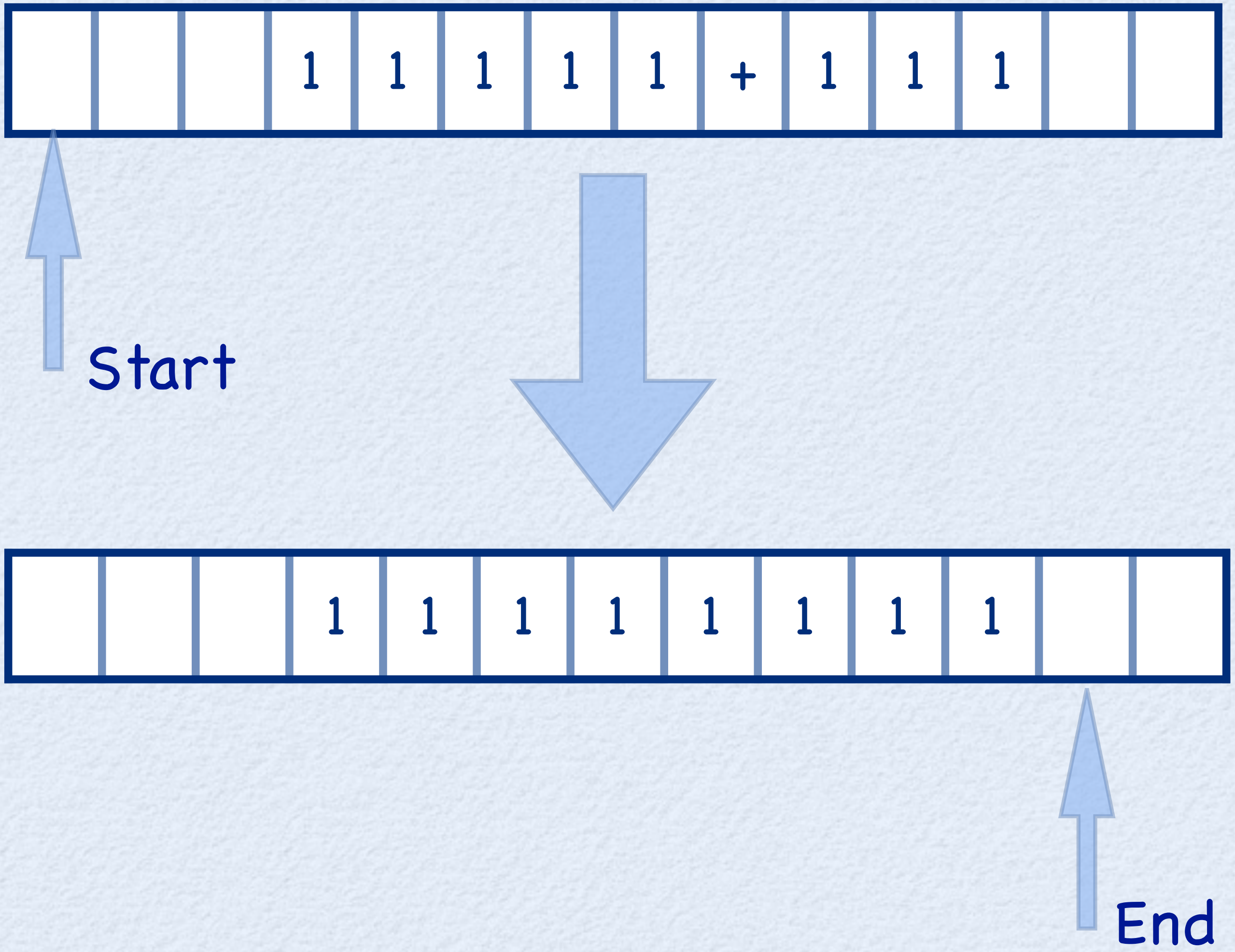
| Q\Γ | " " | "1" | "+" |
|---|---|---|---|
| 1 | 1/" ",R | 2/"1",R | |
| 2 | | 2/"1",R | 3/"+",R |
| 3 | | 4/"+",L | |
| 4 | | | 5/"1",R |
| 5 | 6/" ",L | 4/"+",L | 5/"+",R |
| 6 | | | HALT/" ",R |

# 5+3 on a Turing Machine




| Q\Γ | " " | "1" | "+" |
|---|---|---|---|
| 1 | 1/" ",R | 2/"1",R | |
| 2 | | 2/"1",R | 3/"+",R |
| 3 | | 4/"+",L | |
| 4 | | | 5/"1",R |
| 5 | 6/" ",L | 4/"+",L | 5/"+",R |
| 6 | | | HALT/" ",R |

# 5+3 on a Turing Machine (No Move Allowed)




| State/Input | " " | "1" | "0" | "+" |
|---|---|---|---|---|
| 0 | -/R/- | -/R/1 | | |
| 1 | | -/R/- | | "0"/R/2 |
| 2 | -/L/3 | -/R/4 | | |
| 3 | | | " "/-/stop | |
| 4 | -/L/5 | -/R/4 | | |
| 5 | | -/L/5 | " "/-/stop | |

# The Ackermann Function

- The Ackermann function is a simple example of a computable function that grows much faster than polynomials or exponentials even for small inputs.
- The Ackermann function is defined recursively for non-negative integers m and n as follows:

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

# Ackermann Function Value Table

| 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 | $2^{65536}-3$ | $2^{2^{65536}}-3$ | A(3,A(4,3)) | A(3,A(4,4)) |
| m = 5 | 65533 | A(4,65533) | A(4,A(5,1)) | A(4,A(5,2)) | A(4,A(5,3)) | A(4,A(5,4)) |
| m = 6 | A(4,65533) | 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 Digits

A(4,2) =
20035299304068464649790723515602557504478254755697514192650169737108940595563114530895061308809333481010382343429072631818229493821188126688695063647615470291650418719163515879663472194429309279820843091048559905701593189596395248633723672030029169695921561087649488892540908059114570376752085002066715637023661263597471448071117748158809141357427209671901518362825606180914588526998261414250301233911082736038437678764490432059603791244909057075603140350761625624760318637931264847037437829549756137709816046144133086921181024859591523801953310302921628001605686701056516467505680387415294638422448452925373614425336143737290883037946012747249584148649159306472520151556939226281806916507963810641322753072671439981585088112926289011342377827055674210800700652839633221550778312142885516755540733451072131124273995629827197691500548839052238043570458481979563931578535100189920000241419637068135598404640394721940160695176901561197269823378900 17
...
133663377137843441619405312144529185518013657555866761501937302969193207612000925506508158327550849934076879725236998702356793102680413674571895664143185267905471716996299036301554564509004480278905570196832831363071899769915316667920895876857229060091547291963638167359667395997571032601557192023734858052112811745861006515259888384311451189488055212914577569914657753004138471712457796504817585639507289533753975582208777750607233944558789590571915673 3

# Halting Problem

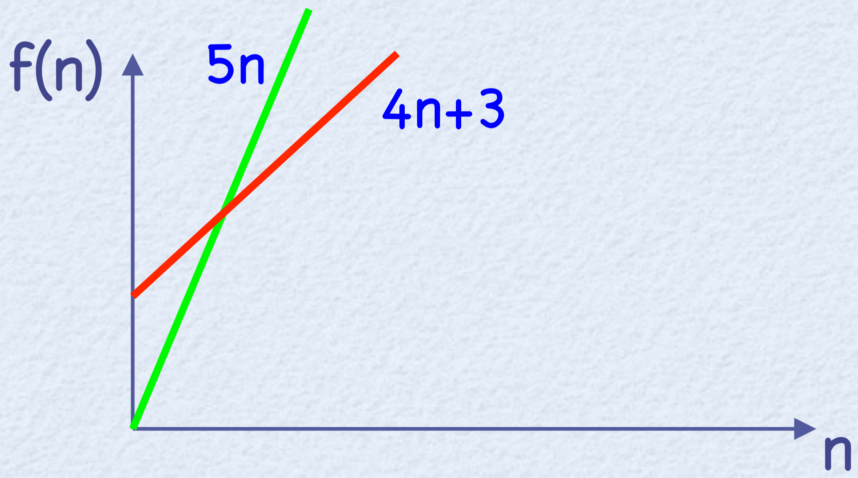
- A problem that cannot be solved by any machine in finite time (or any equivalent formalism) is called uncomputable.
  - An uncomputable problem cannot be solved by any real computer.

**The Halting Problem:**

- Given an arbitrary machine and its input, will the machine eventually halt?
- The Halting Problem is provably uncomputable – which means that it cannot be solved in practice.

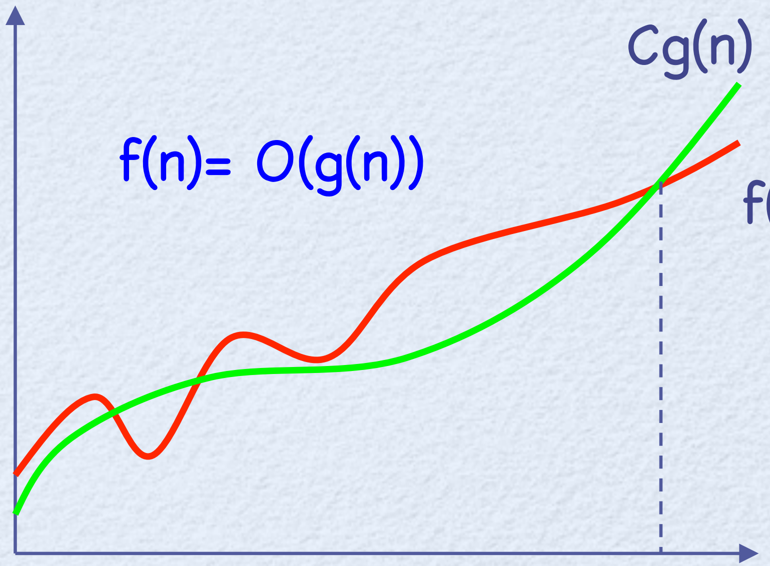
# The Big-Oh

- An algorithm f(n) is O(g(n)), read "has order g(n)", if there exist constants $C > 0$ and integer $n_0$ such that the algorithm f(n) requires at most $C*g(n)$ steps for all input sizes $n \geq n_0$.
- Example:
    - Algorithm A takes $4n + 3$ steps, that is, it is $O(n)$.
    - Choose $C = 5$ and $n_0 = 4$, then $4n + 3 < 5n$ for all $n \geq 4$.



# Facts About Big-Oh

- Big-Oh focuses on growth rate as running time of input size approaches infinity ($n \rightarrow \infty$).
- Big-Oh does not say anything about the running time on small input.
- The function $g(n)$ in $O(g(n))$ is a simple function for comparison of different algorithms:
  - $1, n, \log n, n \log n, n^2, 2^n, \ldots$



# On Running Time Estimation

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

# Performance Analysis

- **Best-Case (Lower Bound):**
  - The search for a given element A in an array of size n can be O(1), if the element is the first. (Also applies to binary search trees.)
- **Worst-Case (Upper Bound):**
  - The search for a given element A in an array of size n is O(n), if the element is the last in the array. (For binary search trees, it is also O(n), if the element is the last in a totally unbalanced binary search tree, but O(log n) for a balanced search tree.)
- **Average-Case:**
  - The search for a given element A in an array of size n takes on average n/2, whereas the lookup in a binary search tree is O(log n).

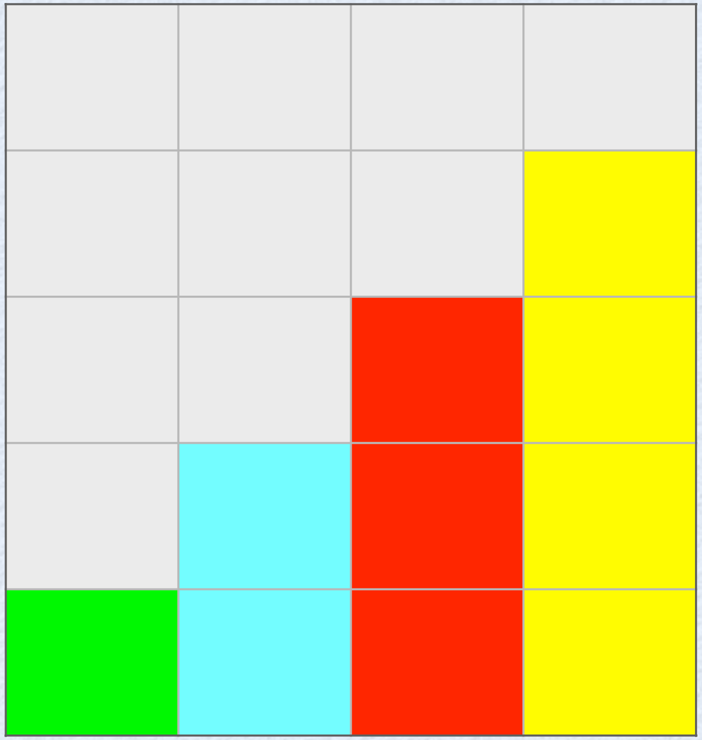
# Constant Time

- 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, in order to keep him quiet for a while. As we know today, this did not work out, since:

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

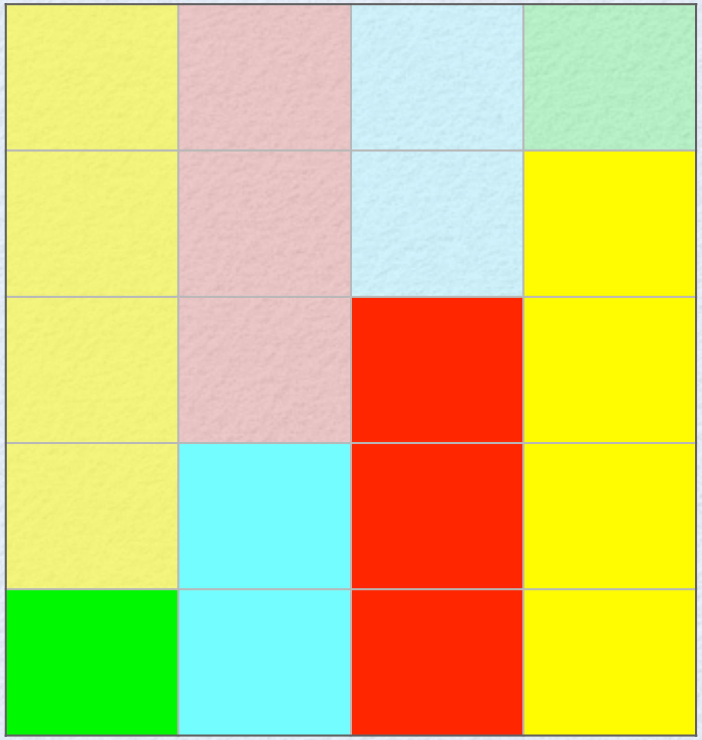
which is O(1).

# How does Gauss's formula work?



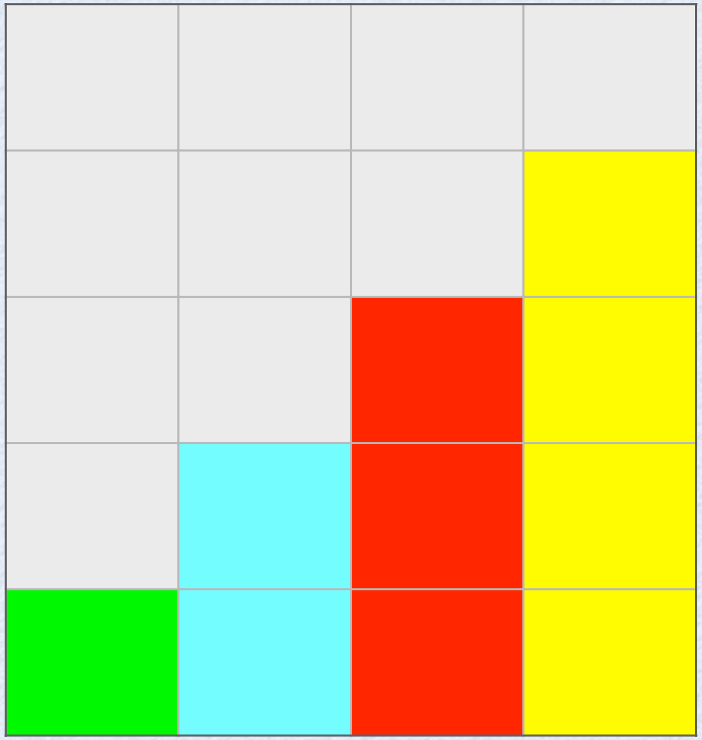
$1 + 2 + 3 + 4 = ?$

# Let's look at the whole rectangle.



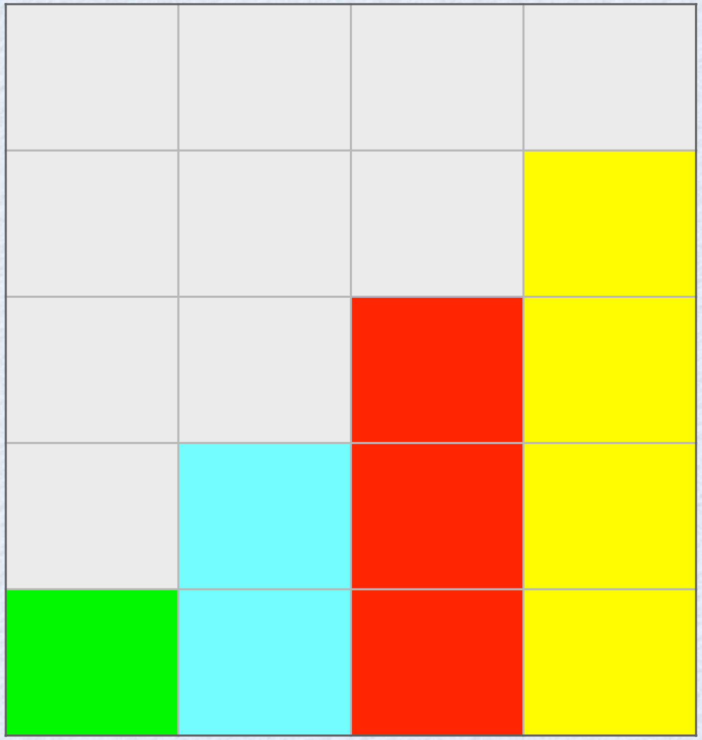
$$2 \times (1 + 2 + 3 + 4) = 4 \times 5$$

# So, we have ...



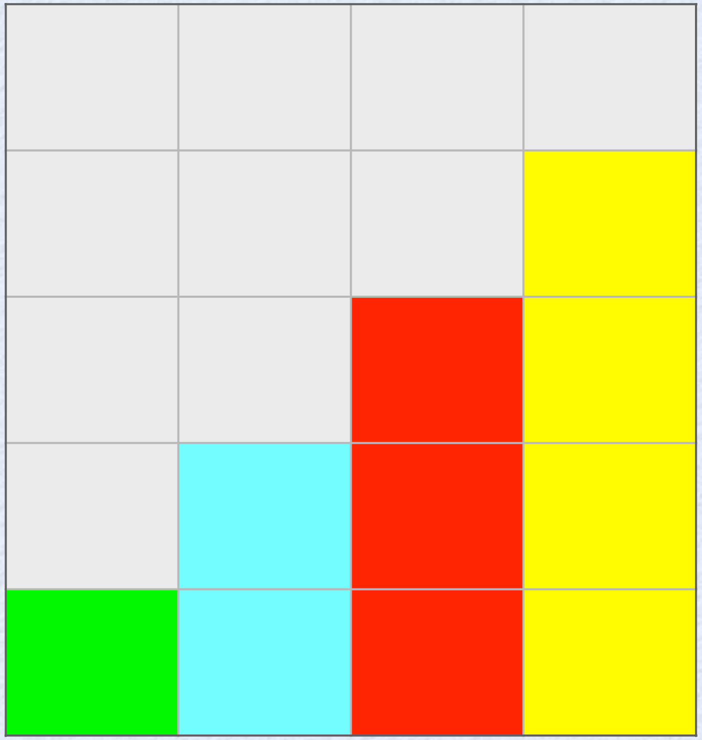
$$1 + 2 + 3 + 4 = (4 \times 5) / 2$$

# We generalize …



$$1 + 2 + \ldots + n = n \times (n + 1) / 2$$

# We focus on the growth rate n → ∞



$n \times (n + 1) / 2$ is $O(1)$

# Polynomial Time

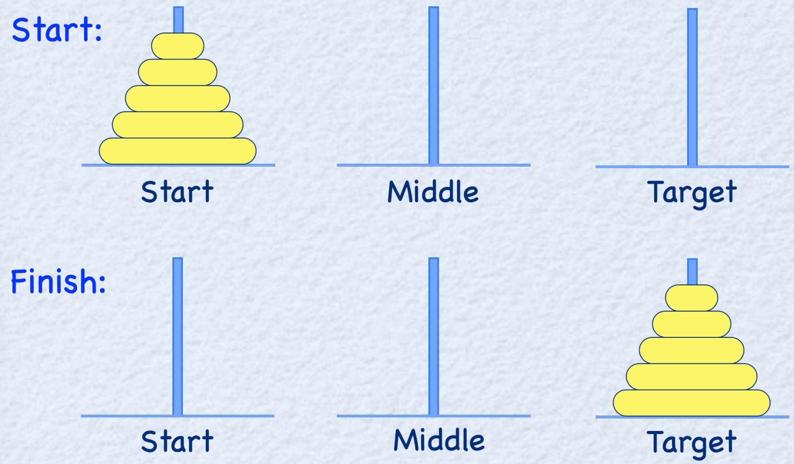
- $4n^2 + 3n + 1$:
  - Ignore lower-order terms: $4n^2$
  - Ignore constant coefficients: $O(n^2)$
- $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 = O(n^k)$

# Logarithmic & Exponential Functions

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

# Towers of Hanoi: The Recursive Procedure

**procedure** TON( n, S, T, M )
**begin**
  **if** n > 0 **then**
    TON( n-1, S, M, T )
    d(T) $\leftarrow$ 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
  - $T(n-1) = 2T(n-2) + 1 = 2*(2T(n-3) + 1) + 1 = 2^2T(n-3) + 2^1 + 1$
  - $T(n-2) = 2T(n-3) + 1 = 2*(2T(n-4) + 1) + 1$
  - …
  - $T(2) = 2T(1) + 1 = 2(2^1 + 1) + 1 = 2^2 + 2^1 + 1$
  - $T(1) = 2T(0) + 1 = 2^1 + 1$
  - $T(0) = 1$
- Solving the recursive equations, we obtain
  - $T(n) = 2^n + 2^{(n-1)} + \ldots\ 2^1 + 1 = 2^{(n+1)} - 1 = O(2^n)$

# A Simple Example $\sum_{i=1}^{n} i^3$

SumI3.cpp

```cpp
2  int sum( int n )
3  {
4      int Result = 0;
5
6      for ( int i = 1; i <= n; i++ )
7          Result += i * i * i;
8
9      return Result;
10 }
```

Line: 13  Column: 1  C++  Tab Size: 4  sum

$\Rightarrow$ $2n + 8 = O(n)$

- We count the computation units in each line:
  - line 2: Zero, the declaration requires no time.
  - line 4 & 9: One time unit each.
  - line 6: Hidden costs for initializing i, testing i <= n, and incrementing i; 1 time unit initialization, n+1 time units for all tests, and n time units for all increments: 2n + 2.
  - line 7: 4 time units, two multiplications, one addition, and one assignment.

# Ranking of Big-Oh

- Fastest: $O(1)$

  $O(\log n)$

  $O(n)$

  $O(n \log n)$

  $O(n^2)$

  $O(n^2 \log n)$

  $O(n^3)$

  $O(2^n)$

- Slowest: $O(n!)$

# General Rules

# For Loops

```
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 statement. Then a for-loop has a running time of at most Cn or O(n).
- For loops have a linear running time.

# Nested For Loops

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

- The running time of a nested for-loop is at most the running time of statement multiplied by the product of the sizes of all the for-loops.
- Let $C$ be the running time of statement. Then a nested for-loop has a running time of at most $Cn*n$ or $O(n^2)$.
- Nested for-loops have quadratic running time. Any additional nesting level adds one factor. A $k$-nested for-loop requires polynomial time or $O(n^k)$.

# Consecutive Statements

$statement_1;$
$statement_2;$
…
$statement_n;$

- The running time for consecutive statements is the sum of each statement.
- Let $m_i$ be the running time for each statement. Then consecutive statements have a running time of at most $m_1 + m_2 + \ldots m_n$ or $O(m)$ where $m = max(m_1, m_2, \ldots, m_n)$.

# If-Then-Else 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 of the running times of the true-statement and the false-statement.
- This can be an overestimate in some cases, but it is never and underestimate.

# Maximum Subsequence Sum Problem in O(n³)

DataProvider.cpp

```cpp
/**
 *  This method computes the maximum sum for a
 *  subsequence of values in the array fNumbers.
 *
 *  @return The maximum subsequence sum in O(n3)
 */
int DataProvider::maxSubSumON3() const
{
    int lMaxSum = 0;

    for ( int i = 0; i < fEntryCount; i++ )
        for ( int j = i; j < fEntryCount; j++ )
        {
            int lThisSum = 0;

            for ( int k = i; k <= j; k++ )
                lThisSum += fNumbers[k];

            if ( lThisSum > lMaxSum )
                lMaxSum = lThisSum;
        }

    return lMaxSum;
}
```

Line: 1    Column: 1    C++    Tab Size: 4

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j} 3$$

# Maximum Subsequence Sum Problem in O(n)

DataProvider.cpp

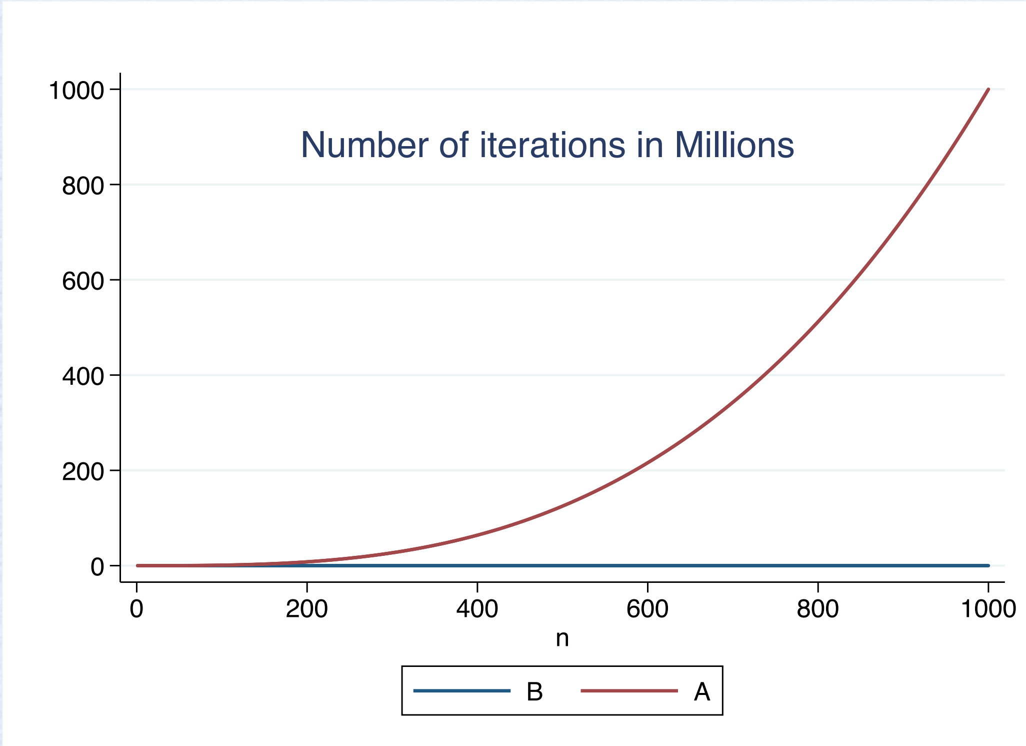
```cpp
/**
 *  This method computes the maximum sum for a
 *  subsequence of values in the array fNumbers.
 *
 *  @return The maximum subsequence sum in O(n)
 */
int DataProvider::maxSubSumON() const
{
    int lMaxSum = 0;
    int lThisSum = 0;

    for ( int i = 0; i < fEntryCount; i++ )
    {
        lThisSum += fNumbers[i];

        if ( lThisSum > lMaxSum )
            lMaxSum = lThisSum;
        else
        {
            if ( lThisSum < 0 )
                lThisSum = 0;
        }
    }

    return lMaxSum;
}
```

Line: 1  Column: 1  C++  Tab Size: 4

$$\sum_{i=0}^{n-1} 3$$

# Running Time T (Algorithm)

- Test environment: MacPro, 2.66 GHz Quad-Core Intel Xeon
- Algorithm A - $O(n^3)$:
  - n = 2,000: T(A) = 5s
  - n = 5,000: T(A) = 72s
  - n = 10,000: T(A) = 579s
- Algorithm B - $O(n)$:
  - n = 2,000: T(B) < 1s
  - n = 5,000: T(B) < 1s
  - n = 10,000: T(B) < 1s



# Algorithmic Patterns

- Direct solution strategies:
  - Brute force and greedy algorithms
- Backtracking strategies:
  - Simple backtracking and branch-and-bound algorithms
- Top-down solution strategies:
  - Divide-and-conquer algorithms
- Bottom-up solution strategies:
  - Dynamic programming
- Randomized strategies:
  - Monte Carlo algorithms

# Brute-force Algorithms

- Brute-force algorithms are not distinguished by there structure.
- Brute-force algorithms are separated by their way of solving problems.
- A problem is viewed as a sequence of decisions to be made. Typically, brute-force algorithms solve problems by exhaustively enumerating all the possibilities.

# Bubble Sort

BubbleSort.cpp

```cpp
void doBubbleSort( int aArray[], unsigned int aLength )
{
    for ( unsigned int i = aLength - 1; i > 0 ; i-- )
    {
        for ( int j = 0; j < i; j++ )
        {
            if ( aArray[j+1] < aArray[j] )
                swap( j+1, j );
        }
    }
}
```

Line: 19 Column: 13 C++ Tab Size: 4 BubbleSort::BubbleSort

- Bubble sort uses a nested for-loop to sort an array in increasing order.
- Bubble sort is $O(n^2)$. It works fine on small arrays, but it is unsuitable on larger data sets.

# Greedy Algorithms

- Greedy algorithms do not really explore all possibilities. They are optimized for a specific attribute.
- Example: Knapsack Problem
  - Profit - Maximal value of items
  - Weight - Maximal weight stored first
  - Density - Maximal profit per weight
- Greedy algorithms produce a feasible solution, but do not guarantee an optimal solution.

# Sudoku Solver: Greedy

1. Find M[i,j] with the minimal number of choices.
2. Let C be the possibilities for M[i,j].
3. Set M[i,j] to first element of C.
4. Solve puzzle recursively with new configuration.
5. If 4. succeeds, then report result and exit.
6. If 4. fails, then set M[i,j] to next element if C. If there is no more element, then report error.
7. Continue with 4.

# Sudoku

A hard puzzle:

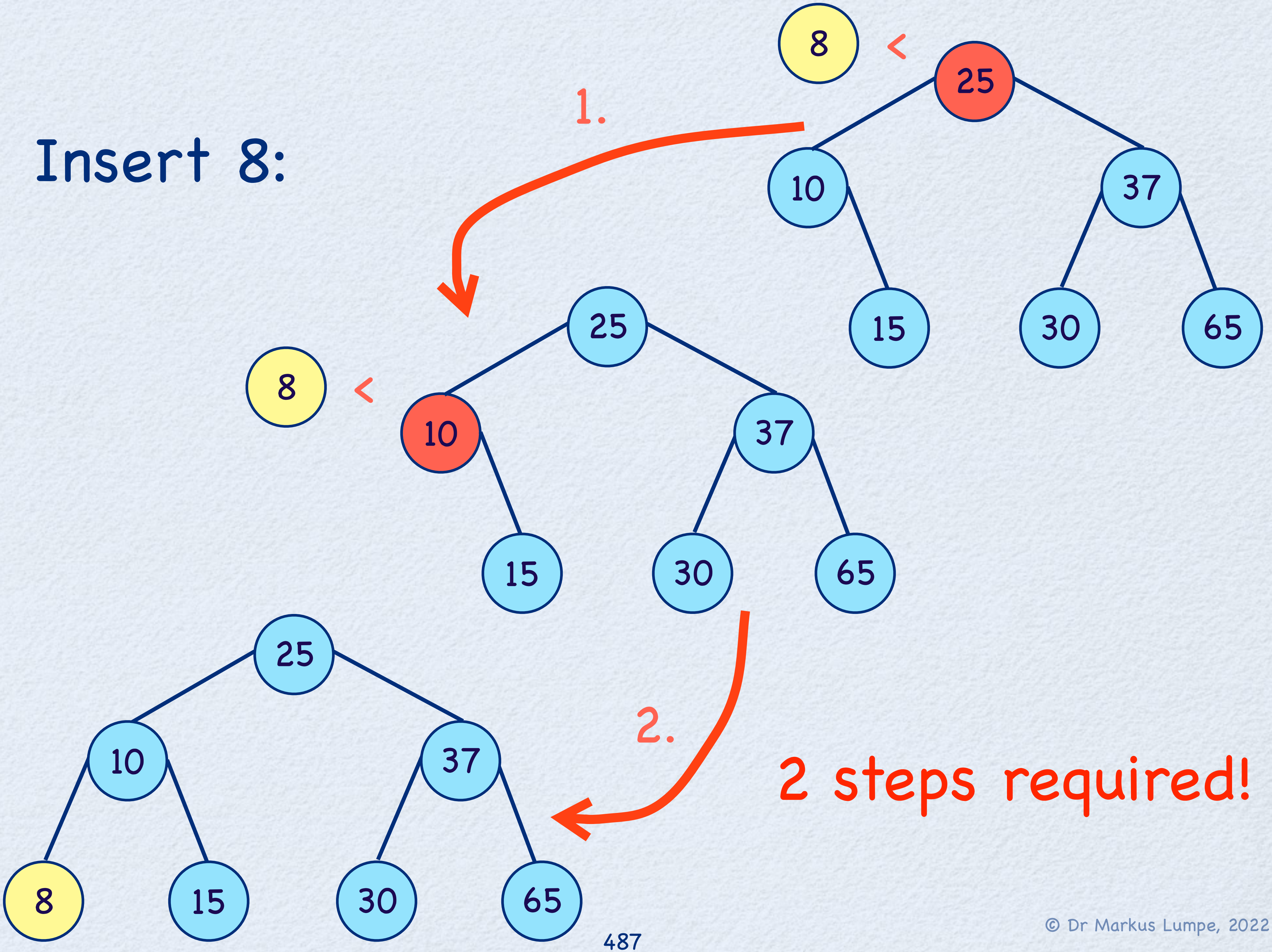
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | | | 8 | | | 7 | 9 | |
| | | | 1 | | | | 3 | |
| | 6 | 9 | | | | 8 | | |
| | 4 | | 6 | 3 | | | | |
| 1 | | 2 | | | | 3 | | 6 |
| | | | | 5 | 1 | | 7 | |
| | | 1 | | | | 9 | 2 | |
| | 2 | | | | 4 | | | |
| | 9 | 6 | | | 3 | | | 8 |

3,5,7,8

# Divide-and-Conquer

- Top-down algorithms use recursion to divide-and-conquer the problem space.
- This class of algorithms has the advantage that not all possibilities have to be explored.
- Example: Binary Search, Merge Sort, Quick Sort

# Insert 8:




2 steps required!

# Binary Search: O(log n)

BinarySearch.cpp

```cpp
int doBinarySearch( int aSortedArray[], int aLength, int aSearchValue )
{
    int lLowIndex = 0;                    // leftmost array index
    int lHighIndex = aLength - 1;         // rightmost array index

    while ( lLowIndex <= lHighIndex )     // at least one more element
    {
        int lMidIndex = (lLowIndex + lHighIndex) / 2;

        if ( aSortedArray[lMidIndex] == aSearchValue )
            return lMidIndex;             // element found

        if ( aSortedArray[lMidIndex] > aSearchValue )
            lHighIndex = lMidIndex - 1; // new rightmost array index
        else
            lLowIndex = lMidIndex + 1;  // new leftmost array index
    }
    return -1;  // element not found
}
```

Line: 38 Column: 1 C++ Tab Size: 4 printArrayLn

Terminal

```
Search 7 in [1, 2, 3, 3, 4, 5, 5, 6, 7, 9]
Search 7 in [5, 5, 6, 7, 9]
Search 7 in [7, 9]
8
Search 20 in [1, 2, 3, 3, 4, 5, 5, 6, 7, 9]
Search 20 in [5, 5, 6, 7, 9]
Search 20 in [7, 9]
Search 20 in [9]
-1
```

log(10) = 2.3

# Backtracking Algorithms

- A backtracking algorithm systematically considers all possible outcomes for each decision.
- Backtracking algorithms are distinguished by the way in which the space of possible solutions is explored. Sometimes a backtracking algorithm can detect that an exhaustive search is unnecessary.
- Example: Knapsack Problem

# Sudoko Solver: Backtracking

1. Find M[i,j] with the minimal number of choices.
2. Let C be the possibilities for M[i,j].
3. Set M[i,j] to first element of C.
4. Solve puzzle recursively with new configuration.
5. If 4. succeeds, then report result and exit.
6. If 4. fails, then set M[i,j] to next element if C. If there is no more element, then report error.
7. Continue with 4.

# Bottom-up

- Bottom-up algorithms employ dynamic programming.
- Bottom-up algorithms solve a problem by solving a series of subproblems.
- These subproblems are carefully devised in such a way that each subsequent solution is obtained by combining the solutions to one or more of the subproblems that have already been solved.
- Example: Parsing, Pretty-Printing

# Fast Fibonacci

IterFibo.cpp

```cpp
long iterFibo( long n )
{
    long prev = 1;
    long curr = 0;

    for ( long i = 1; i <= n; i++ )
    {
        long next = curr + prev;
        prev = curr;
        curr = next;
    }

    return curr;
}
```

Line: 3 Column: 1 C++ Tab Size: 4

- The iterative solution of the Fibonacci function has its origin in dynamic programming.
- We define the Fibonacci sequence as a table with two elements: the previous value and the current value. In each step we compute the next value by adding the table entries.

# Randomized Algorithms

- Randomized algorithms behave randomly.
- Randomized algorithms select elements in an random order to solve a given problem.
- Eventually, all possibilities are explored, but different runs can produce results faster or slower, if a solution exists.
- Example: Monte Carlo Methods, Simulation

# Sorting by Fisher&Yates

FisherAndYatesSort.cpp

```cpp
void sortByFisherAndYates( int aArray[], int aLength )
{
    bool isSorted;

    do
    {
        isSorted = true;
        for ( int i = 0; isSorted && i < aLength-1; i++ )
            if ( aArray[i] > aArray[i+1] )
                isSorted = false;

        if ( !isSorted )
            shuffle( aArray, aLength ); // by Fisher&Yates
    } while (!isSorted);
}
```

Line: 41 Column: 26 C++ Tab Size: 4 printArray

- There is at least one configuration that satisfies the sorting criterion, but the use of the Fisher&Yates shuffling process makes this algorithm $O(n*n!)$. It is called "Bogosort."