Data Structures and Algorithms 2

Prof. Ahmed Guessoum
National Higher School of AI

Chapter 2

Algorithm Complexity

Chapter Outline

- Algorithms efficiency
- Algorithm analysis:
  - Machine-dependent vs Machine-independent
- Function ordering
  - Weak Order; Big-Oh (asymptotically $\leq$); Big-Ω (asymptotically $\geq$); Big-θ (asymptotically =); Little-oh (asymptotically <).
- Algorithm complexity analysis
  - Rules for complexity analysis
  - Analysis of 4 algorithms for the max subsequence sum
- Master Theorem

Design challenges

- Designing an algorithmic solution is (almost) good:
  - The algorithm should not take “ages”
  - It should not consume “too much” memory.
- In this chapter, we will discuss the following:
  - How to estimate the time required for a program.
  - How to reduce the running time of a program from days or years to fractions of a second.
  - The results of careless use of recursion.
  - Very efficient algorithms to raise a number to a power and to compute the greatest common divisor of two numbers.

Algorithms Efficiency

- A city has *n* view points
- Buses move from one view point to another
- A bus driver wishes to follow the shortest path (travel time wise).
- Every view point is connected to another by a road.
- However, some roads are less congested than others.
- Also, roads are one-way, i.e., the road from view point 1 to 2, is different from that from view point 2 to 1.

- How to find the shortest path between any two pairs?
- Naïve approach:
  - List all the paths between a given pair of view points
  - Compute the travel time for each.
  - Choose the shortest one.
- How many paths are there between any two view points (without revisits)?

$$\mathbf{n! \cong (n/e)^n}$$

It will be impossible to run the algorithm for n = 30 (roughly 11^{30} paths)

What is efficiency of an algorithm?

- Run time in the computer is machine-dependent
- Example: Multiply two positive integers a and b
- Subroutine 1: Multiply a and b
- Subroutine 2:

```
v = a,    w =  b
While   w > 1
      { v  = v + a;
        w = w - 1}
Output v
```

Machine-Dependent Analysis

- First subroutine has 1 multiplication.
- Second subroutine has *b* additions and *b* subtractions.
- For some architectures, 1 multiplication is more expensive than *b* additions and *b* subtractions.
- Ideally:
  - programme all alternative algorithms
  - run them on the target machine
  - find which is more/most efficient!

Machine-Independent Analysis

- We will assume that every basic operation takes constant time
- Example of Basic Operations:
  – Addition, Subtraction, Multiplication, Memory Access
- Non-basic Operations:
  – Sorting, Searching
- Efficiency of an algorithm is thus measured in terms of the number of basic operations it performs
  – We do not distinguish between the basic operations.

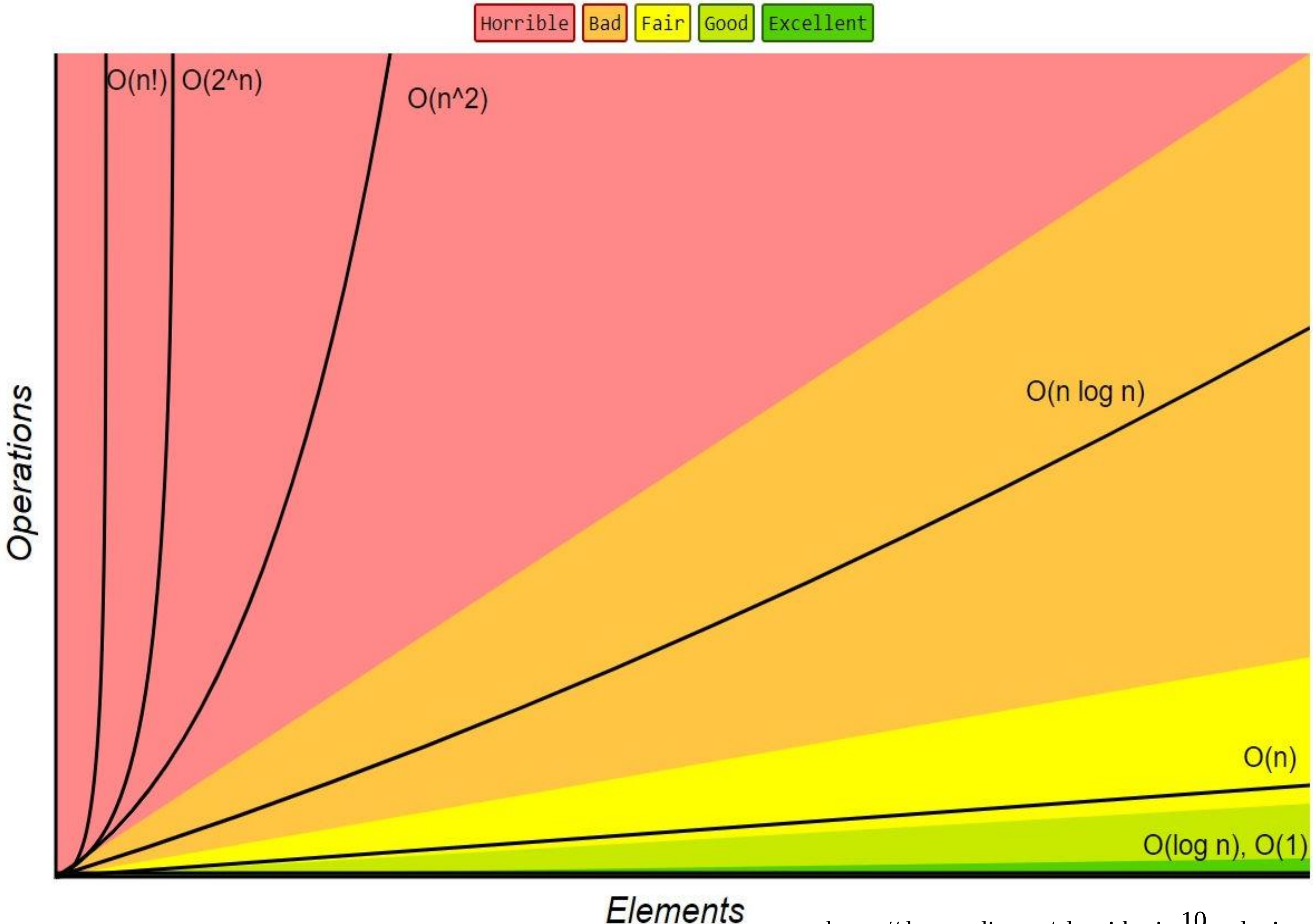
- Subroutine 1 uses 1 basic operation
- Subroutine 2 uses $2b$ basic operations

  ➔ Subroutine 1 is more efficient.

- This measure is good for all large input sizes

- **In fact**, we will not worry about the exact number of operations, but will look at ``broad classes' of values.
  - Let there be n inputs.
  - If an algorithm needs n basic operations and another needs 2n basic operations, we will consider them to be in the same efficiency category.
  - However, we distinguish between exp(n), n, log(n)

- We worry about algorithms speed for large input sizes (i.e. rate of growth of algorithm speed based on input size)



https://devopedia.org/algorithmic-complexity

Weak ordering

- In this chapter, f and g are functions from the set of natural numbers to itself.
- Consider the following definitions:
  - We will consider two functions to be equivalent, $f \sim g$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ where $0 < c < \infty$
  - We will state that $f < g$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- For the functions we are interested in, these define a weak ordering.

Weak ordering

Let $f(n)$ and $g(n)$ describe the run-time of two algorithms. In general, there are functions s.t.:

- If $f(n) \sim g(n)$, then it is always possible to improve the performance of one function over the other by purchasing a faster computer.
- If $f(n) < g(n)$, then you can <u>never</u> purchase a computer fast enough so that the second function **always** runs in less time than the first.

- Note that for small values of n, it may be reasonable to use an algorithm that is asymptotically more expensive, but we will consider these on a one-to-one basis.

Function Orders: Big Oh Notation

Definition 2.1

$f(n) = O(g(n))$ if there are positive *constants* c and n_0 such that $0 \le f(n) \le c * g(n)$ when $n \ge n_0$

i.e. A function $f(n)$ is $O(g(n))$ if the rate of growth of $f(n)$ is not faster than that of $g(n)$.

i.e. if $lim_{n\to\infty} f(n)/g(n)$ exists and is finite, then $f(n)$ is $O(g(n))$

Intuitively, (not exactly) $f(n)$ is $O(g(n))$ means $f(n) \le g(n)$ for all n beyond some value n_0; i.e. $g(n)$ is an *upper bound* for $f(n)$.

Remark

Before we begin, let us make some assumptions:

- Our functions will describe the time or memory required to solve a problem of size n
- Restrictions are put on the functions:
  - They are defined for $n \geq 0$
  - They are strictly positive for all n
    - In fact, $\mathrm{f}(n) > c$ for some value $c > 0$
    - That is, any problem requires at least one instruction and byte
  - They are increasing (monotonic increase)

Examples of Functions

sqrt(n) , n, 2n, ln (n), exp(n), n + sqrt(n) , n + n^2

$\lim_{n\to\propto}$ sqrt(n) /n = 0,	sqrt(n) is O(n)
$\lim_{n\to\propto}$ n/sqrt(n) = infinity,	n is not O(sqrt(n))
$\lim_{n\to\propto}$ n /2n = 1/2,	n is O(2n)
$\lim_{n\to\propto}$ 2n /n = 2,	2n is O(n)

Other examples:

$\lim_{n \to \propto} \ln(n) / n = 0$	$\ln(n)$ is $O(n)$
$\lim_{n \to \propto} n / \ln(n) =$ infinity	n is not $O(\ln(n))$
$\lim_{n \to \propto} \exp(n) / n =$ infinity	$\exp(n)$ is not $O(n)$
$\lim_{n \to \propto} n / \exp(n) = 0$	n is $O(\exp(n))$
$\lim_{n \to \propto} (n + \mathrm{sqrt}(n)) / n = 1$	$n + \mathrm{sqrt}(n)$ is $O(n)$
$\lim_{n \to \propto} n / (\mathrm{sqrt}(n) + n) = 1$	n is $O(n + \mathrm{sqrt}(n))$
$\lim_{n \to \propto} (n + n^2) / n =$ infinity	$n + n^2$ is not $O(n)$
$\lim_{n \to \propto} n / (n + n^2) = 0$	n is $O(n + n^2)$

Implication of Big Oh notation

- Suppose we know that our algorithm uses at most $O(f(n))$ basic steps for any n inputs, and n is sufficiently large,
  - then we know that our algorithm will terminate after executing at most $f(n)$ basic steps.
- We also know that a basic step takes a constant time on a machine.

  ➔ Our algorithm will terminate in at most $f(n)$ units of time, for all large n.

Other Complexity: Big Ω Notation

Now a lower bound notation, $\Omega(n)$

Definition 2.2

$f(n) = \Omega(g(n))$ if there are positive *constants* c and n_0 such that $f(n) \geq c * g(n)$ when $n \geq n_0$.

i.e. $\lim_{n\to\infty} (f(n)/g(n)) > 0$, if $\lim_{n\to\infty} (f(n)/g(n))$ exists.

We say *g(n)* is a *lower bound* on *f(n)*, i.e. no matter what specific inputs we have, <u>the algorithm will not run faster than this lower bound</u>.

Interpretation of the Ω Notation

- Suppose, an algorithm has complexity Ω(f(n)) .
- This means that there exists a positive constant c such that for all sufficiently large n, there exists at least one input for which the algorithm consumes at least c * f(n) steps.
- Big-O vs. Big-Omega:

➢ **Big-Ω** notation: gives a lower bound of the running time of an algorithm. Big-Ω(n) means the algorithms runs at least in *n* time but could actually take a lot longer.

➢ **Big-O** notation: gives an upper bound so O(n) would mean the algorithm runs in its worst case in *n* (i.e. linear) time.

Other Complexity: Big θ Notation

Definition 2.3

$f(n) = \theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

- $\theta(g(n))$ is referred to as ``asymptotic equality"
- $lim_{n \to \infty}(f(n)/g(n))$ is a finite, positive constant, if it exists.
- $f(n)$ has **a rate of growth** equal to that of $g(n)$

Other Complexity: Little-oh Notation

Definition 2.4:

$f(n) = o(g(n))$ if, for all positive constants c, there exists an n_0 such that $f(n) < c * g(n)$ when $n > n_0$. (``asymptotic strict inequality'')

- Less formally, $f(n) = o(g(n))$ if $f(n) = O(g(n))$ and $f(n) \neq \theta(g(n))$.
- $f(n)$ is $o(g(n))$ if given any positive constant c, there exists some m such that $f(n) < c * g(n)$ for all $n \geq m$

  $lim_{n \to \propto} (f(n)/g(n)) = 0$, if $lim_{n \to \propto} (f(n)/g(n))$ exists

- Little *oh* means: "is ultimately smaller than"

Rates of gowth

Suppose that $\mathrm{f}(n)$ and $\mathrm{g}(n)$ satisfy $\lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} = c$

If $\lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} = c$ where $0 < c < \infty$, it follows that $\mathrm{f}(n) = \boldsymbol{\Theta}(\mathrm{g}(n))$

If $\lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} = c$ where $0 \le c < \infty$, it follows that $\mathrm{f}(n) = \mathbf{O}(\mathrm{g}(n))$

If $\lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} = 0$, we will say $\mathrm{f}(n) = \mathbf{o}(\mathrm{g}(n))$, i.e. $\mathrm{f}(n)$ has a rate of growth less than that of $\mathrm{g}(n)$

Terminology

Asymptotically less than or equal to	**O**
Asymptotically greater than or equal to	**Ω**
Asymptotically equal to	**θ**
Asymptotically strictly less	**o**

Recap

$$f(n) = \mathbf{o}(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = \mathbf{O}(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) = \mathbf{\Theta}(g(n)) \qquad 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) = \mathbf{\Omega}(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

Example Functions

sqrt(n) , n, 2n, ln n, exp(n), n + sqrt(n) , n + n^2

$\lim_{n\to\infty}$ sqrt(n) /n = 0 sqrt(n) is o(n), O(n)

$\lim_{n\to\infty}$ n/sqrt(n) = infinity n is Ω(sqrt(n))

$\lim_{n\to\infty}$ n/ln(n) = infinity n is Ω(ln(n))

$\lim_{n\to\infty}$ 2n /n = 2 2n is θ(n)

$\lim_{n\to\infty}$ n /2n = 1/2 n is θ(2n)

Terminology

The most common classes are given names:

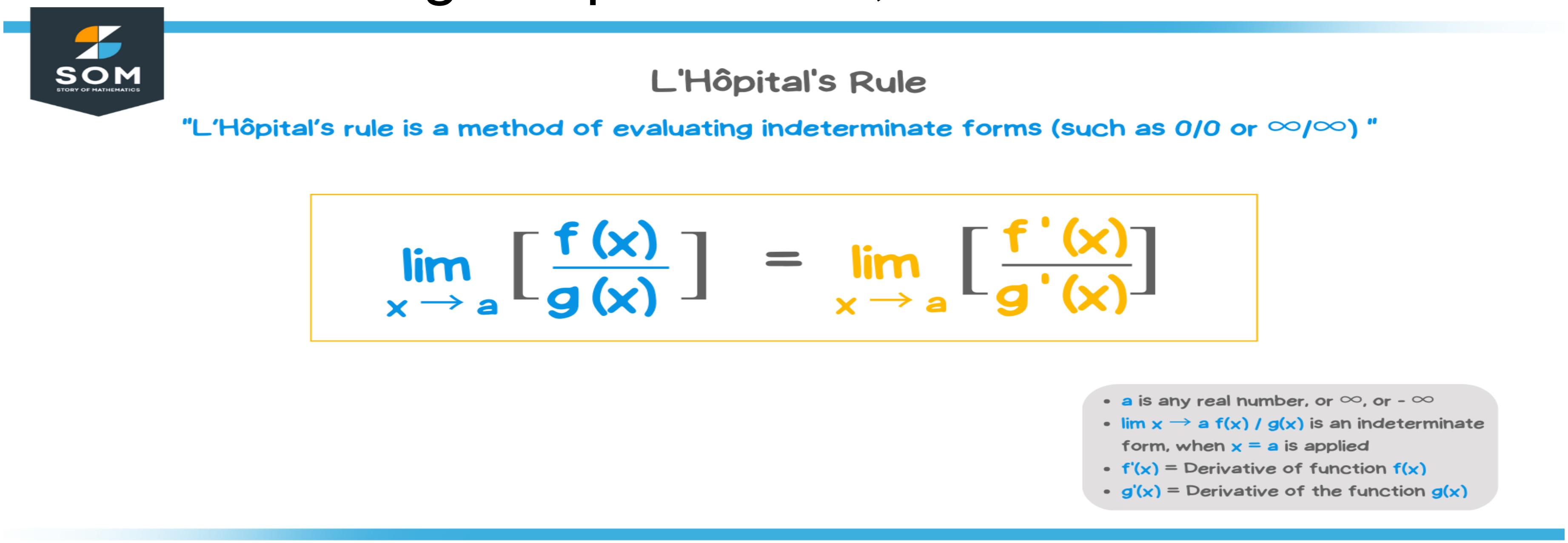
$\Theta(1)$	constant
$\Theta(\ln(n))$	logarithmic
$\Theta(n)$	linear
$\Theta(n \ln(n))$	"n log n"
$\Theta(n^2)$	quadratic
$\Theta(n^3)$	cubic
$2^n, e^n, 4^n, \ldots$	exponential

Little-o as a Weak Ordering

- We can show that, for example

$$\ln(n) = \mathbf{o}(n^p) \qquad \text{for any } p > 0$$

- Proof: Using l'Hôpital's rule,



- So we have:

$$\lim_{n\to\infty} \frac{\ln(n)}{n^p} = \lim_{n\to\infty} \frac{1/n}{pn^{p-1}} = \lim_{n\to\infty} \frac{1}{pn^p} = \frac{1}{p} \lim_{n\to\infty} n^{-p} = 0$$

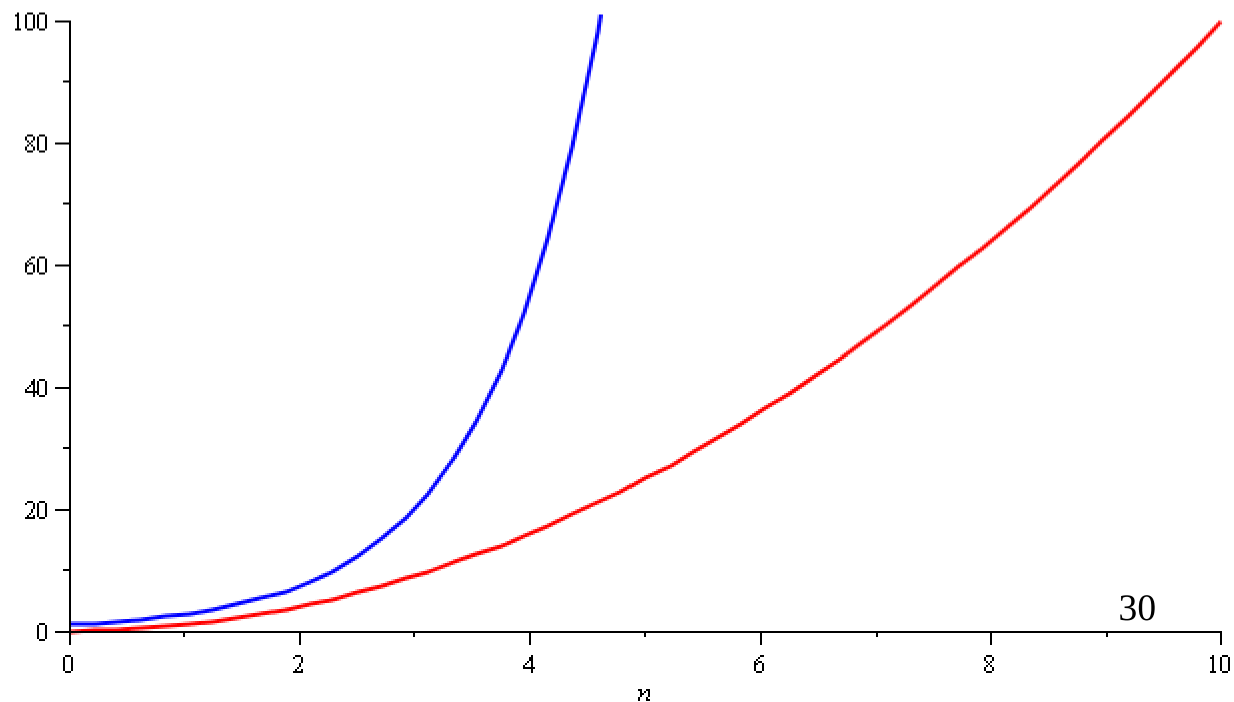
Algorithms Analysis

- An algorithm is said to have *polynomial time complexity* if its run-time may be described by $O(n^d)$ for some fixed $d \geq 0$.
  - We will consider such algorithms to be **efficient.**
- Problems that have no known polynomial-time algorithms are said to be intractable.
  - Traveling salesman problem: find the shortest path that visits n cities
  - Best run time: $\Theta(n^2 2^n)$

Algorithm Analysis

In general, you don't want to implement exponential-time or exponential-memory algorithms

– Warning: don't call a quadratic curve "exponential".



Rules for arithmetic with big-O symbols

- If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, then
  - (a) $T_1(n) + T_2(n) = O(f(n) + g(n))$ (intuitively and less formally it is $O(\max(f(n), g(n)))$),
  - (b) $T_1(n) * T_2(n) = O(f(n) * g(n))$.
- If $T(n)$ is a polynomial of degree k, then $T(n) = \theta(n^k)$.
- $log^k\, n = O(n)$ for any constant k.

  This tells us that logarithms grow very slowly.

Rules for arithmetic with big-O symbols

- If $f(n) = O(g(n))$, then $c * f(n) = O(g(n))$ for any constant c.
- If $f_1(n) = O(g(n))$ but $f_2(n) = o(g(n))$, then $f_1(n) + f_2(n) = O(g(n))$.
- If $f(n) = O(g(n))$, and $g(n) = o(h(n))$, then $f(n) = o(h(n))$. (complexity of $g \circ h$)
- These are not all of the rules, but they're enough for most purposes.

Complexity of a Problem vs Complexity of an Algorithm

A problem is $O(f(n))$ means there is some $O(f(n))$ algorithm to solve the problem.

A problem is $\Omega(f(n))$ means every algorithm that can solve the problem is $\Omega(f(n))$

Algorithm Complexity Analysis

- We define $T\text{avg}(N)$ and $T\text{worst}(N)$, as the average and worst-case running time, resp., used by an algorithm on input of size N. Clearly, $T\text{avg}(N) \leq T\text{worst}(N)$.
- Occasionally, the best-case performance of an algorithm is analyzed.
  - but of little interest: does not represent the typical behavior.
- Average-case performance often reflects typical behavior
- **Worst-case performance represents a guarantee for performance on any possible input**
- We are interested in algorithm analysis not programme analysis: implementation issues/details/inefficiencies, etc.

Algorithm Complexity Analysis

Consider the following algorithm

```
diff = sum = 0;
For (k=0: k < N; k++)
      sum → sum + 1;
      diff → diff - 1;
For (k=0: k < 3N; k++)
      sum → sum - 1;
```

- First line takes 2 basic steps
- Every iteration of first loop takes 2 basic steps.
- First loop runs N times
- Every iteration of second loop takes 1 basic step
- Second loop runs for 3N times
- Overall, 2 + 2N + 3N steps
- This is O(N)

Rules for Complexity Analysis

Complexity of a loop:

O(Number of iterations in a loop * maximum complexity of each iteration)

Nested Loops:

Analyze the innermost loop first,

Complexity of next outer loop =

number of iterations in this loop * complexity of inner loop, etc.

```
sum = 0;
For (i=0; i < N; i++)
    For (j=0; j < N; j++)   sum → sum + 1;
```

Inner loop: O(N)

Outer loop: N iterations

Overall: $O(N^2)$

```
for( i = 0; i < n; ++i )
    a[ i ] = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < n; ++j )
        a[ i ] += a[ j ] + i + j;
```

First loop: O(N)

Inner loop: O(N)

Outer loop: N iterations

Overall: $O(N^2) + O(N)$
$\rightarrow O(N^2)$

If (Condition)

S1

Else S2

Maximum of the two complexities

If (yes)

Algo 1

else Ago 2

O(Algo 1)

Analysis of recursion

- Suppose we have the code (not a good one):

```
Long fib (int n) {
        if (n == 0 || n == 1)                    1
                return 1;                        2
        else
                return fib(n – 1) + fib(n – 2);  3
}
```

T(0) = T(1) = 1;

n>=2 T(n) = cost of constant op at 1 + cost of line 3 work

T(n) = 1 op + (addition + 2 function calls) = O(1) + (addition + cost of fib(n-1) + cost fib(n-2)) = 2 + T(n-1) + T(n-2)

Analysis of recursion

- Easy to show by induction T(n) >= fib(n)
- Can prove for n>4, fib(n) >= $(3/2)^n$

➔Running time of the programme grows exponentially

By using an array and a for loop, the programme running time can be reduced substantially.

<u>Ex:</u> Do it and analyse your algorithm.

Maximum Subsequence Problem

Given an array of N elements

Need to find i, j such that the sum of all elements between the ith and jth positions is maximum for all such sums

Running time of 4 algorithms for max subsequence sum (in seconds) [Weiss, Fig 2.2]

	Algorithm Time			
Input Size	1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 100$	0.000159	0.000006	0.000005	0.000002
$N = 1{,}000$	0.095857	0.000371	0.000060	0.000022
$N = 10{,}000$	86.67	0.033322	0.000619	0.000222
$N = 100{,}000$	NA	3.33	0.006700	0.002205
$N = 1{,}000{,}000$	NA	NA	0.074870	0.022711

Algorithm 1

```
/**
 * Cubic maximum contiguous subsequence sum algorithm.
 */
int maxSubSum1( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); ++i )
        for( int j = i; j < a.size( ); ++j )
        {
            int thisSum = 0;

            for( int k = i; k <= j; ++k )
                thisSum += a[ k ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }

    return maxSum;
}
```

Analysis of Algorithm 1

Precise analysis is obtained from the sum

$\sum_{i=0}^{N-1} \sum_{j=i}^{N-1} \sum_{k=i}^{j} 1$ # of times

thisSum += a[k]; # is executed

1. First: $\sum_{k=i}^{j} 1 = j - i + 1$
2. Inner loop: # sum of first N-i elements

$$\sum_{j=i}^{N-1} (j-i+1) = (N-i+1)(N-i)/2$$

3. Outer Loop:

$$\sum_{i=0}^{N-1} (N-i+1)(N-i)/2 = (N^3 + 3N^2 + 2N)/6$$

Analysis of Algorithm 1

Overall: $O(N^3)$

Note: The innermost loop can be made more efficient leading to $O(N^2)$

Algorithm 2

```
/**
 * Quadratic maximum contiguous subsequence sum algorithm.
 */
int maxSubSum2( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); ++i )
    {
        int thisSum = 0;
        for( int j = i; j < a.size( ); ++j )
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }
    }

    return maxSum;
}
```

Divide and Conquer

- Break a big problem into two small sub-problems
- Solve each of them efficiently.
- Combine the two solutions

Maximum subsequence sum by divide and conquer

- Divide the array into two parts: left part, right part each to be solved recursively
- Max. subsequence lies completely in left, or completely in right or spans the middle.
- If it spans the middle, then it includes the max subsequence in the left ending at the last element and the max subsequence in the right starting from the center

4 –3 5 –2 -1 2 6 -2

Max subsequence sum for first half = 6

second half = 8

Max subsequence sum for first half ending at the last element (4^{th} elements included) is 4

Max subsequence sum for second half starting at the first element (5^{th} element included) is 7

Max subsequence sum spanning the middle is 4 + 7 = 11

Max subsequence spans the middle

Algorithm 3: Divide & Conquer

```
int maxSumRec( const vector<int> & a, int left, int right )
{
    if( left == right )  // Base case
        if( a[ left ] > 0 )
            return a[ left ];
        else
            return 0;

    int center = ( left + right ) / 2;
    int maxLeftSum  = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    int maxLeftBorderSum = 0, leftBorderSum = 0;
    for( int i = center; i >= left; --i )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }
```

```
    int maxRightBorderSum = 0, rightBorderSum = 0;
    for( int j = center + 1; j <= right; ++j )
    {
        rightBorderSum += a[ j ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                    maxLeftBorderSum + maxRightBorderSum );
}

/**
 * Driver for divide-and-conquer maximum contiguous
 * subsequence sum algorithm.
 */
int maxSubSum3( const vector<int> & a )
{
    return maxSumRec( a, 0, a.size( ) - 1 );
}
```

Algorithm 3 Analysis

- If N=1; lines 8 to 12 executed; taken to be one unit
  ➔ $T(1) = 1$
- N>1: 2 recursive calls + 2 for loops + some

  bookkeeping ops (e.g. lines 14, 34)
  - The 2 for loops (lines 19 to 32): clearly $O(N)$
  - Lines 8, 14, 18, 26, 34: constant time; ignored compared to $O(N)$
  - Recursive calls made on half the array size each.

    ➔ $2 * T(N/2)$

SO: programme time is $2 * T(N/2) + O(N)$ with $T(1) = 1$

Complexity Analysis

$T(1) = 1$

$T(n) = 2T(n/2) + c.n$

$= 2.c.n/2 + 4T(n/4) + c.n$

$= 4T(n/4) + 2c.n$

$= 8T(n/8) + 3c.n$

=…………..

$= 2^i T(n/2^i) + i.c.n$

=………………… (reach a point when $n = 2^i$ $i = \log n$

$= n.T(1) + c\, n \log n$

$n + c\, n \log n =$ **$O(n \log n)$**

Algorithm 4

```
/**
 * Linear-time maximum contiguous subsequence sum algorithm.
 */
int maxSubSum4( const vector<int> & a )
{
    int maxSum = 0, thisSum = 0;

    for( int j = 0; j < a.size( ); ++j )
    {
        thisSum += a[ j ];

        if( thisSum > maxSum )
            maxSum = thisSum;
        else if( thisSum < 0 )
            thisSum = 0;
    }

    return maxSum;
}
```

Algorithm 4 Analysis

- $T(N) = O(N)$ Obvious!
- What is not obvious at first sight is the logic of the algorithm.

Exercise: convince yourself that the 4 algorithms do the required work. (Trace them!)

Binary Search

- You have a sorted list of numbers
- You need to search the list for a specific number
- If the number exists
  - Then find its position
  - Else detect that
- In Binary search, subdivide the list into 2
- If number is in the center, done;
  else: **double-recursively** search into the left one and into the **right** one.

```
Search(num, A[],left, right)
  {
        if (left == right)
         {
              if (A[left ] == num)     return(left) and exit;
              else conclude NOT PRESENT and exit;
          }
      center = ⌊(left + right)/2⌋;
       If (A[center] < num)
          Search(num, A[], center + 1, right);
      If (A[center] > num)
        Search(num, A[], left, center );
     If (A[center] == num) return(center) and exit;
}
```

Complexity Analysis

$T(n) = T(n/2) + c$

Similar logic as for the analysis of Algorithm 3 above ➔

O(log n) complexity

Master Theorem

- **Master Theorem**: used to calculate time complexity of divide-and-conquer algorithms.
- Simple and quick way to calculate the time complexity of a recursive relation.
- It applies to recurrence relations of the form:
  $T(n) = aT(n/b) + f(n)$

where

- n is the size of the input;
- a is the number of subproblems in the recursion;
- n/b is the size of each subproblem (all assumed to have the same size);
- f(n): cost of work done outside recursive calls; includes cost of dividing the problem and solutions merge cost.

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

This slide and the following two are taken from:
Su, J. CS 161 Lecture 3, University of Stanford,
/https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture3.pdf , (Retrieved 02/10/2023)

Examples of Master Theorem

- <u>Example 1</u>: $T(n) = 9\,T(n/3) + n$.
  Here $a = 9$, $b = 3$, $f(n) = n$, and $n^{\log_b a} = n^{\log_3 9} = \theta(n^2)$.

Since $f(n) = O(n^{\log_3 9-\varepsilon})$ for $\varepsilon = 1$, case 1 of the Master Theorem applies, so $T(n) = \theta(n^2)$.

- <u>Example 2</u>: $T(n) = T(2n/3)+1$. Here $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^0 = 1$.
  Since $f(n) = \theta(n^{\log_b a})$, case 2 of the Master Theorem applies, so $T(n) = \theta(\log n)$.

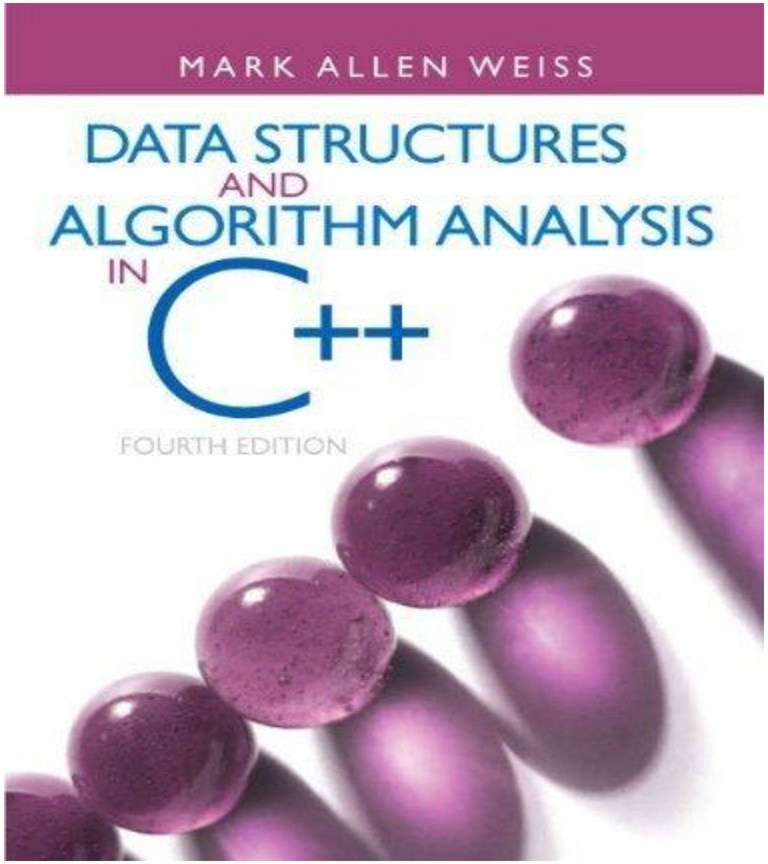
- Example 3: $T(n) = 3T(n/4) + n \log n$. Here $n^{\log b\ a} = n^{\log_4 3} = O(n^{0.793})$. For $\varepsilon = 0.2$, we have $f(n) = \boldsymbol{\Omega}(n^{\log_4 3+\varepsilon})$.

So case 3 applies if we can show that $a\ f(n/b) \leq c\ f(n)$ for some $c < 1$ and all sufficiently large n. This would mean $3\ n/4 \log(n/4) \leq c\ n \log n$. Setting $c= 3/4$ would cause this condition to be satisfied. Hence $T(n) = \theta(n \log n)$.

- Example 4: $T(n) = 2T(n/2)+n \log n$. Here the Master Theorem does not apply.

$n^{\log_b a} = n^{\log_2 2} = n$, and $f(n) = n \log n$. Case 3 does not apply because $f(n)$ is not $\boldsymbol{\Omega}(n^{\log_b a+\varepsilon})$ since $\lim (f(n) / n^{\log_b a+\varepsilon})$ tends towards 0 when n tends towards infinity.

Slides based on the textbook



Mark Allen Weiss, (2014) Data Structures and Algorithm Analysis in C++, 4th edition, Pearson.

Acknowledgement: This **course PowerPoints** make substantial (non-exclusive) use of the PPT chapters prepared by Prof. Saswati Sarkar from the University of Pennsylvania, USA, themselves developed on the basis of the course textbook. Other references, if any, will be mentioned wherever applicable.