COMP 6651 / Winter 2022
Dr. B. Jaumard

Lecture 1: Mathematical Preliminaries - Order of Growth, Recurrence Relations

January 7, 2022

Outline

Prerequisites

- Prerequisites
  - COMP 5361: Discrete Structures and Formal Languages
  - COMP 5511: Principles of Data Structures
- You are expected to have a **good working knowledge** of the material you were taught in those courses:
  - sets, relations, functions, logic, proof techniques,
  - graph theory,
  - counting techniques, permutations and combinations,
  - binary search trees, hashing, stacks and queues, sorting.
- You are expected to be able to write programs using either C++ or Java
- It will be difficult to follow or appreciate this course without a proper background preparation.

Getting Started

- Computer code = set of instructions forming a computer program
  - data structure(s) + language + algorithm(s)
- Data structures organize information
  - Examples: stacks, lists, arrays, trees, ...
- Algorithms manipulate information.

- Algorithm: a set of well-defined steps for solving a problem.
- It must be
  - finite,
  - each step is precisely defined,
  - the order of steps is precisely defined,
  - it must terminate for any input.
- Examples: sorting algorithms, search algorithms

Do no confuse a program and an algorithm:
A program is an implementation of an algorithm to be run on a specific computer and operating system.
An algorithm is more abstract - it does not deal with machine specific details - think of it as a method for solving a problem

- Aim of the course
  - study additional basic **types** of efficient algorithms.
  - study how to analyze algorithms
  - study how to deal with problems for which we have no fast algorithms
  - study how to design an efficient algorithm
- Measures of efficiency
  - time
  - space
  - both are needed for execution of the algorithm as function of the size of input.

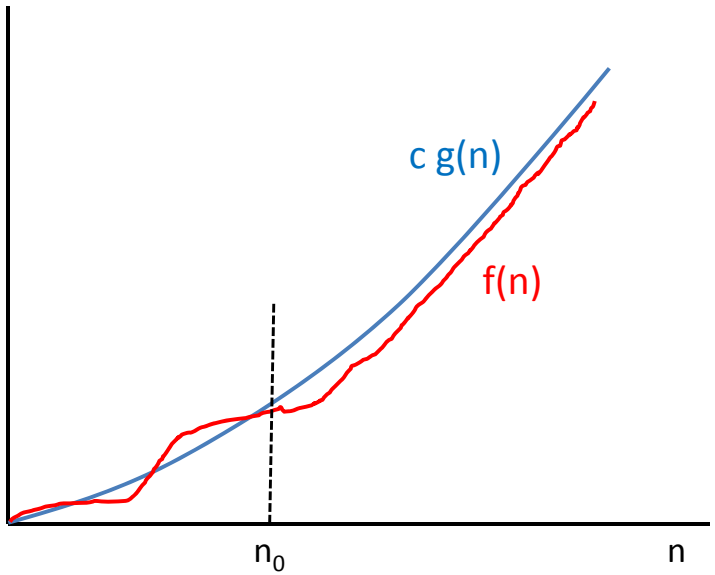
- Examples
  - Sort n items: if all items are approximately of the same fixed size, n is a measure of the input size.
  - Find whether an integer i is a prime number: input size is the number of digits in i.
- Algorithm Analysis: Predict the resources that the algorithm requires.
- Empirical Methods: Run experiments and measure the time and space.
- Analytical Methods: Analyze the structure of an algorithm and derive, using mathematical methods, the time and space needed.
- We will focus on analytical methods.

Order of growth: O notation (1/2)

The counting of time is not exact, so we express the run-time using the asymptotic notation

O-notation: $f(n) = O(g(n))$
if there exists positive constants c, n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

- $f(n)$ grows at most as fast as $g(n)$
- $g(n)$ is an upper bound on the growth of $f(n)$



Order of growth: O notation (2/2)

- $O(1)$ describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.
- When using $O()$ notation, drop constants and low-order terms. Why? When the problem size gets sufficiently large, those terms don't matter.
- Two algorithms can have the same $O()$ time complexity even though one is always faster than the other,
  - If algorithm 1 requires n^2 operations and algorithm 2 requires $10 \times n^2 + n$ operations. Time complexity is $O(n^2)$ for both algorithms, but algorithm 1 will always be faster than algorithm 2.

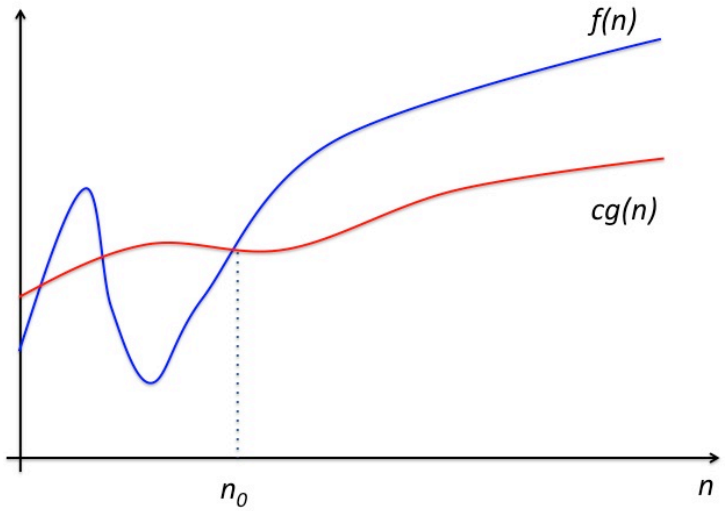
  Constants and low-order terms do matter in terms of which algorithm is actually faster.
- Constants do not matter in terms of the question of how an algorithm "scales" (i.e., how does the algorithm's time change when the problem size doubles).

Order of growth: Ω notation

The counting of time is not exact, so we express the run-time using the asymptotic notation

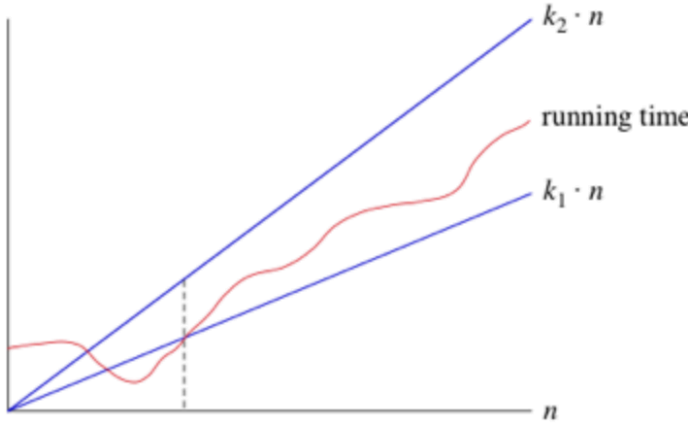
Ω-notation: $f(n) = \Omega(g(n))$ if there exists positive constants c, n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.

- $f(n)$ grows at least as fast as $g(n)$
- $g(n)$ is a lower bound on the growth of $f(n)$



Order of growth: Θ and o notations

- Θ-notation: $f(n) = \Theta(g(n))$
  if there exists positive constants k_1, k_2, n_0 such that
  $0 \le k_1 g(n) \le f(n) \le k_2 g(n)$ for all $n \ge n_0$.
  - $f(n)$ grows exactly as fast as $g(n)$



- o-notation: $f(n) = o(g(n))$
  if for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0$.
  - $f(n)$ grows slower than $g(n)$
  - $2n = o(n^2)$, but $n^2 \ne o(n^2)$
  - Within the scope of the course,

$$f(n) = o(g(n)) \quad \text{if and only if (by definition)} \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Examples

- $2n^2+5n-6 = O(2^n)$
- $2n^2+5n-6 = O(n^3)$
- $2n^2+5n-6 = O(n^2)$
- $2n^2+5n-6 \neq O(n)$
- $2n^2+5n-6 \neq \Theta(2^n)$
- $2n^2+5n-6 \neq \Theta(n^3)$
- $2n^2+5n-6 = \Theta(n^2)$
- $2n^2+5n-6 \neq \Theta(n)$

Examples (con't)

- $2n^2+5n-6 \neq \Omega(2^n)$
- $2n^2+5n-6 \neq \Omega(n^3)$
- $2n^2+5n-6 = \Omega(n^2)$
- $2n^2+5n-6 = \Omega(n)$
- $2n^2+5n-6 = o(2^n)$
- $2n^2+5n-6 = o(n^3)$
- $2n^2+5n-6 \neq o(n^2)$
- $2n^2+5n-6 \neq o(n)$

Operations

- c is a constant:
  - $O(f(n)+c) = O(f(n))$
  - $O(cf(n)) = O(f(n))$
- If $f_1(n) = O(f_2(n))$ then
  - $O(f_1(n)+f_2(n)) = O(f_2(n))$
- If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then
  - $O(f_1(n) \times f_2(n)) = O(g_1(n) \times g_2(n))$

Using limits

- If $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$
  - then $g(n) = O(f(n))$ and $f(n) = \Omega(g(n))$
- If $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$
  - then $g(n) = \Omega(f(n))$ and $f(n) = O(g(n))$
- If $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ where c is a nonzero constant
  - then $f(n) = \Theta(g(n))$ and $g(n) = \Theta(f(n))$

- Worst case analysis: The longest running time (space) for any input of length n.
- Average case analysis (expected): The running time (space) averaged over all inputs of length n.
  - This is the most useful information, but often difficult to get.
  - It also may involve assumptions on whether all case are occurring equally often, etc.
- Best case analysis: The shortest running time (space) for any input of length n.

Analysis of algorithms often gives a run-time in form of a recurrence relation: $T(n) = T(\lfloor n/2 \rfloor) + c$

Binary search

Let A a sorted array of n elements $\hookrightarrow$ is x in the list?

Binary search

IF the array has one element THEN
compare x to that element and return the appropriate answer
ELSE
get the element M in the 'middle' of the array;
IF $x \geq M$ THEN
call the algorithm recursively on the 'second half' of the array
ELSE
call the algorithm recursively on the 'first half' of the array.

Suppose that the number of elements is $n = 2^k$

Required number of comparisons using binary search (d_n) is defined by :

$$d_1 = 1$$

$$d_n = d_{n/2} + 1 \qquad (n \geq 2)$$

How to solve recurrence equations

- Classical three step solution:

Step 1. Recursive substitution
Step 2. Guess the form of the solution
Step 3. Use mathematical induction to show that the solution works

- In the next slides: A general method to solve "most" cases of linear recurrence equations, using characteristic equation

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + ... + c_k a_{n-k} + f(n)$$

- The master theorem

Linear Recurrence Relations

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + ... + c_k a_{n-k} + f(n)$$

- Linear: first-order polynomial in a_{n-1}, ...,a_{n-k}
- Degree: k
- Constant coefficients: c_i coefficients are constant
- Homogeneous equation if $f(n) = 0$
- Inhomogeneous equation if $f(n) \neq 0$

Examples

	Degree	Linear	Homogeneous	Constant coefficients
$a_n = a_{n-1} + a_{n-2}$	2	Yes	Yes	Yes
$a_n = 2a_{n-1}$	1	Yes	Yes	Yes
$a_n = a_{n-1} + a_{n-2} + 2^{n-2}$	2	Yes	No	Yes
$a_n = a_{n-1} + a_{n-2}^2$	2	No	-	-
$a_n = na_{n-1} + n$	1	Yes	No	No

How to solve linear homogeneous, constant coefficients, order k equations

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + ... + c_k a_{n-k} \qquad \text{with } k \text{ initial conditions}$$

Step 1 : Set the characteristic equation

$$r^k - c_1 r^{k-1} - c_2 r^{k-2} - ... - c_k = 0$$

Step 2 : Solve the characteristic equation, obtain its roots

Step 3 : If all roots are unique

- then $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n + ... + \alpha_m r_m^n$ where $r_1, r_2, ..., r_m$ are the distinct roots
- else we have t distinct roots: r_i with multiplicity m_i and

$$\begin{aligned} a_n = &(\alpha_{10} + \alpha_{11} n + \alpha_{12} n^2 + \cdots + \alpha_{1,m_1-1} n^{m_1-1}) r_1^n \\ &+ (\alpha_{20} + \alpha_{21} n + \alpha_{22} n^2 + \cdots + \alpha_{2,m_2-1} n^{m_2-1}) r_2^n \\ &+ ... + (\alpha_{t0} + \alpha_{t1} n + \alpha_{t2} n^2 + \cdots + \alpha_{t,m_t-1} n^{m_t-1}) r_t^n \end{aligned}$$

Step 4 : Use the initial conditions to get α_i or α_{ij}

Example 1: $a_0 = 2, a_1 = 7, a_n = a_{n-1} + 2a_{n-2} \quad n \geq 2$

- Characteristic equation: $r^2 = r + 2$
- Two distinct roots: $r_1 = 2, r_2 = -1$
- General form of the solution: $a_n = \alpha_1 2^n + \alpha_2 (-1)^n$
- α_1 and α_2 satisfy:

$$\begin{cases} \alpha_1 \times 2^0 + \alpha_2 \times (-1)^0 = 2 & (n = 0) \\ \alpha_1 \times 2^1 + \alpha_2 \times (-1)^1 = 7 & (n = 1) \end{cases}$$

$$\Leftrightarrow \quad \begin{cases} \alpha_1 + \alpha_2 = 2 & (n = 0) \\ 2\alpha_1 - \alpha_2 = 7 & (n = 1) \end{cases} \quad \Rightarrow \quad \alpha_1 = 3 \quad \text{and} \quad \alpha_2 = -1$$

- Solution is $\{a_n\}$ where $a_n = 3 \times 2^n - (-1)^n$

Example 2: 4 initial conditions (values of a_0, a_1, a_2 and a_3)

$$a_n = 2a_{n-1} - 2a_{n-3} + a_{n-4} \quad n \geq 4$$

- Characteristic equation: $r^4 = 2r^3 - 2r + 1$. Observe that:

  $r^4 - 2r^3 + 2r - 1 = 0$ is equivalent to $(r-1)^3(r+1) = 0$.

- One root of multiplicity 1: $r_1 = -1$
  One root of multiplicity 3: $r_2 = 1$
- General form of the solution:

  $$a_n = 1^n \times (\alpha_0 + \alpha_1 n + \alpha_2 n^2) + \alpha_3(-1)^n = \alpha_0 + \alpha_1 n + \alpha_2 n^2 + \alpha_3(-1)^n$$

- ...

Example 3: 3 initial conditions (values of a_1, a_2 and a_3)

$$a_n = a_{n-1} - a_{n-2} + a_{n-3} \quad n \geq 4$$

- Characteristic equation: $r^3 = r^2 - r + 1$. Observe that:

  $r^3 - r^2 + r - 1 = 0$ is equivalent to $(r-i)(r+i)(r-1) = 0$.

- One root of multiplicity 1: $r_1 = 1$
  One root of multiplicity 1: $r_2 = i$ (complex number)
  One root of multiplicity 1: $r_3 = -i$ (complex number)
- General form of the solution: $a_n = \alpha_0 (i)^n + \alpha_1 (-i)^n + \alpha_2 1^n$
- Compute the values of α_0, α_1 and α_2 using the initial conditions. Note that, if the solution of the recurrence relation corresponds to the computing time estimation of an algorithm, the final answer will be an integer (complex parts will canceled out)

How to solve non homogeneous, constant coefficients, order r equations

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + ... + c_k a_{n-k} + f(n) \qquad \text{with } k \text{ initial conditions}$$

Step1 : Solve the associated homogeneous recurrence equation (i.e., without $f(n)$), get the generic solution $\{a_n^{(h)}\}$
Do not apply the step with the initial conditions

Step 2 : Find a particular solution $\{a_n^{(p)}\}$

Step 3 : Overall solution $a_n = a_n^{(h)} + a_n^{(p)}$

Step 4 : Use the initial conditions to get the values of the constants arising in $\{a_n^{(h)}\}$.

Searching for particular solutions (1/2)

$f(n)$ is a polynomial $p(n)$

- search $a_n^{(p)}$ under the form of a polynomial $p(n)$ such that

$$\text{degree}(p(n)) = \text{degree}(f(n)) + t$$

where t is the multiplicity of 1, if 1 is a root of the characteristic equation. $t = 0$ if 1 is not a root of the characteristic equation.

$f(n) = \beta^n$

- search $a_n^{(p)}$ under the form: $Cn^k\beta^n$ where
  - C is a constant
  - if β is a root of the characteristic equation
    - then k is the multiplicity of β,
    - else $k = 0$ with the convention $n^0 = 1$

Searching for particular solutions (2/2)

$f(n)$ is of the form $p(n)+\beta^n$ where $p(n)$ is a polynomial

- Search for a particular solution of the form $f_1(n)+f_2(n)$ where, for each $i \in \{1,2\}$, $f_i(n)$ is a particular solution to the non-homogeneous recurrence

$$\mathbf{a_n} = c_1\mathbf{a_{n-1}} + c_2\mathbf{a_{n-2}} + \ldots + c_k\mathbf{a_{n-k}} + f_i(n)$$

with $f_1(n) = p(n)$ and $f_2(n) = \beta^n$.

$f(n) = p(n)\beta^n$ where $p(n)$ is a polynomial of degree r and β is a constant

- **If β is not a root** of the characteristic equation, then there is a particular solution of the form $q(n)\beta^n$, where q is a polynomial of degree at most r.
- **If β is a root** of the characteristic equation of multiplicity t, then there is a particular solution of the form $q(n) \times n^t\beta^n$, where q is a polynomial of degree at most r.

Recurrence relations vs. divide and conquer algorithms

let A a sorted array of n elements $\hookrightarrow$ is x in the list?

- Straightforward sequential search: compare x with $A[1],...,A[n]$
  $\hookrightarrow$ n comparisons in the worst case, i.e, $O(n)$.
- Divide and conquer search: $O(\log n)$

Binary search

IF the array has one element THEN
compare x to that element and return the appropriate answer
ELSE
get the element M in the 'middle' of the array;
IF $x \geq M$ THEN
call the algorithm recursively on the 'second half' of the array
ELSE
call the algorithm recursively on the 'first half' of the array.

Suppose that the number of elements $n = 2^k$

The required number of comparisons using the binary search, d_n is defined by:

$$d_1 = 1$$

$$d_n = d_{n/2} + 1 \qquad (n \geq 2)$$

How to solve it ?

$$d_1 = 1 \quad \text{and} \quad d_n = d_{n/2} + 1, \qquad (n \geq 2)$$

Nonlinear recurrence equation !
Linearized using a change of variable: $n = 2^k \quad \hookrightarrow \quad a_k = d_{2^k}$

$$a_k = a_{k-1} + 1 \qquad k \geq 1, \qquad a_0 = 1$$

This is a linear non homogeneous recurrence equation with $f(n) = 1$

1a. Homogeneous part: $a_k - a_{k-1} = 0$

1b. Characteristic Equation: $r - 1 = 0$, $r = 1$

1c. $a_k^{(h)} = C_1 (r)^k = C_1 (1)^k = C_1$ is the generic solution for the homogeneous part

2a. A particular solution: $a_k^{(p)} = Bk + C$

2b. Constant values of the particular solution: $a_k^{(p)}$ must satisfy: $a_k^{(p)} = a_{k-1}^{(p)} + 1$,i.e., $Bk + C = B(k-1) + C + 1 \quad \Longrightarrow \quad B = 1$, we take $C = 0$. $a_k^{(p)} = k$

3. $a_k = a_k^{(h)} + a_k^{(p)} = k + C_1$

4a. Satisfying initial condition $a_0 = 1 = 0 + C_1$, $C_1 = 1$.

4b. Finally $a_k = k + 1$.

4c. $d_{2^k} = k + 1$, recall that $n = 2^k$, i.e, $k = \log_2 n$

4d. $d_n = \log_2 n + 1 = O(\log n)$

Some Justifications

Step 1. General solution of the homogeneous part $a_k^{(h)}$: wait for Step 4 before computing the values of the constants

Step 2. Particular solution of the non homogeneous equation

- Search for a polynomial of degree 1, as the "$f(k)$" term is a polynomial of degree 0, and 1 is a root of the characteristic equation with multiplicity 1
- Any value of C is valid, therefore we select the value leading to the simplest expression of $a_k^{(p)}$

Step 3. Superposition of the two solutions: $a_k^{(h)} + a_k^{(p)}$

Step 4. We need first to compute the constant values using the initial conditions, and come back to the variables of the initial equation.

The master theorem for divide and conquer recurrence relations

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 n/b can be $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

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

The master theorem: A particular case

Theorem

Let $a \geq 1$ and $b \geq 2$ be integer constants, let c and k be positive constants, and let $T(n)$ be defined by the recurrence

$$T(n) = aT(n/b) + n^k.$$

Then $T(n)$ can be bounded asymptotically as follows:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

Using the master theorem

Theorem

Let $a \geq 1$ and $b \geq 2$ be integer constants, let c and k be positive constants, and let $T(n)$ be defined by the recurrence

$$T(n) = aT(n/b) + cn^k.$$

Then $T(n)$ can be bounded asymptotically as follows:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

Example 1

$$T_1(n) = 7T_1(n/2) + n^2$$

$$T_1(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.81})$$

Remember that: $7^{\log_2 n} = n^{\log_2 7}$

Example 2

$$T_2(n) = 7T_2(n/3) + n^2$$

$$T_2(n) = \Theta(n^2)$$

Divide-and-conquer algorithms

- Divide the problem into subproblems
- Conquer the subproblem by solving them recursively (small size subproblems are solved directly)
- Combine the solution to the subproblems into a solution of the original problem
- how many subproblems?
- what are the “small” sizes that can be solved directly?
- how to combine solutions of subproblems into a solution of the original problem?

Analyzing the run-time of divide-and-conquer algorithms

Consider a problem of size n.
In most cases, when a subproblem is of size $\leq c$, it takes a constant time: $T(n) = \Theta(1)$ if $n \leq c$
Assume $n > c$, and that the problem can be divided into a instances of the same problem of size $1/b$ of the original size.

- There can be some cost involved in breaking a problem into subproblem: $D(n)$
- There can be some cost involved in combining solutions of subproblems into a solution of the problem: $C(n)$
- Complexity: $T(n) = aT(n/b) + D(n) + C(n)$ if $n > c$

Example: Binary Search

BinSearch (x, A, i, j)

// search where x is in array
// between $A[i]$ and $A[j]$

IF $(i > j)$ return -1;
$\text{mid} \longleftarrow (i+j)/2$;
IF $(x < A[\text{mid}])$ return
BinSearch$(x, A, i, \text{mid}-1)$;
ELSE IF (x > A[mid]) return
BinSearch$(x, A, \text{mid}+1, j)$;
ELSE return mid;

$a = 1,\ b = 2$

cost for breaking a problem into subproblem: $D(n) = \Theta(1)$

cost for combining solutions of subproblems into a solution of the problem: $C(n) = \Theta(1)$

$$T(n) = T(n/2) + C(n) \quad \text{if } n \geq 1$$

$$T(n) = \Theta(\log n)$$

Another Example: MergeSort

Merge-Sort(A, p, r)

// sort $A[p]$ to $A[r]$

IF (p < r);
$\text{mid} \longleftarrow (p+r)/2$;
Merge-Sort (A, p, mid);
Merge-Sort (A, mid, r);
Merge (A, p, mid, r);

$a = 2,\ b = 2$

cost for breaking a problem into subproblem: $D(n) = \Theta(1)$

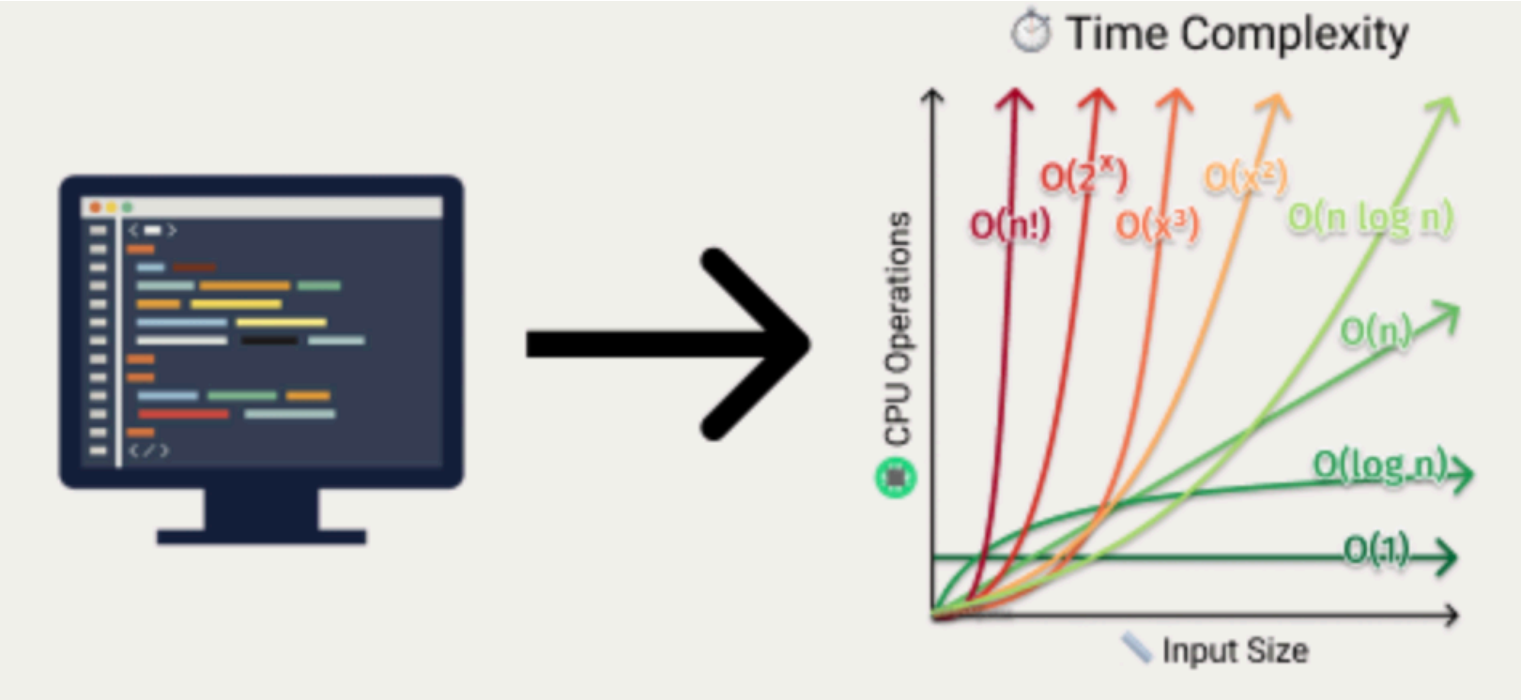
cost for combining solutions of subproblems into a solution of the problem: $C(n) = \Theta(n)$

$$T(n) = 2T(n/2) + cn \quad \text{if } n \geq 1$$

$$T(n) = \Theta(n \log n)$$

How to find time complexity of an algorithm?

From algorithm to big O notation



Sequence of Statements

Sequence of statements
statement 1;
statement 2;
...
statement k;

To compute the total runtime $\rightsquigarrow$ add the times for all statements:
Total runtime = time(statement 1) + time(statement 2)
+ ... + time(statement k)

If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: $O(1)$.

if-then-else statements

```
if (condition) {
   sequence 1 of statements
}
else {
   sequence 2 of statements
}
```

Here, either sequence 1 will execute, or sequence 2 will execute. Therefore, the worst-case time is the slowest of the two possibilities:
max{ time(sequence 1), time(sequence 2)} .

For example, if sequence 1 is $O(N)$ and sequence 2 is $O(1)$ the worst-case time for the whole if-then-else statement would be $O(N)$.

Loops

```
for (i = 1 to N) {
        sequence of statements
}
```

The loop executes N times, so the sequence of statements also executes N times.

If we assume the statements are $O(1)$, the total time for the for loop is $N \times O(1)$, which is $O(N)$ overall.

Nested Loops: # iterations of the inner loop is independent of the value of the outer loop's index

```
for (i = 0 to N) {
    for (j = 0 to M) {
        sequence of statements
    }
}
```

The outer loop executes N times.
Every time the outer loop executes, the inner loop executes M times
$\rightsquigarrow$ the statements in the inner loop execute a total of $N \times M$ times.
Thus, the complexity is $O(N \times M)$.
If the stopping condition of the inner loop is $j < N$ instead of $j < M$ (i.e., the inner loop also executes N times), the total complexity for the two loops is $O(N^2)$.

Nested Loops: # iterations of the inner loop depends on the value of the outer loop's index

```
for (i = 0 to N – 1) {
    for (j = i + 1 to N – 1) {
        sequence of statements
    }
}
```

Value of i	# iterations of inner loop
0	N
1	$N-1$
2	$N-2$
...	...
$N-2$	2
$N-1$	1

Total number of times the sequence of statements executes:
$N + N - 1 + N - 2 + ... + 3 + 2 + 1 \rightsquigarrow O(N^2)$.

Example 1 - Complexity Analysis

Consider the following recurrence:

$$T(n) = 2T(\sqrt{n}) + 1 \quad (1)$$
$$T(1) = 1. \quad (2)$$

Which of the following is true?

1. $T(n) = O(\log \log n)$
2. $T(n) = O(\log n)$ ✓
3. $T(n) = O(\sqrt{n})$
4. $T(n) = O(n)$

Example 2 - Complexity Analysis

Suppose we want to arrange the n numbers stored in any array such that all negative values occur before all positive ones. What is the minimum number of exchanges required in the worst case?

1. $n-1$
2. n
3. $n+1$
4. $n/2$ ✓

- Material covered in the first part of the slides corresponds to Chapter 1 (Section 1.1 to 1.3) of Goodrich and Tamassia (2015) or Chapter I 3 of Cormen *et al.*, 3rd edition (2009).
- A good reference for recurrence solutions (examples and explanations of the solution techniques):
  Townsend, M., *Discrete Mathematics: Applied Combinatorics and Graph Theory*, Benjamin/Cummings, 1987, Chapter 4.
- Some suggestions for exercises (all in Goodrich and Tamassia (2015)) in order to help you mastering what has been presented in Lecture 1: R-1.7 (p. 42), R-1.11 to R-1.15 (p. 43), C-1.8 (p. 45), C-1.10 (p. 45).
- The list of exercises with their solution (course pack available at the bookstore): 1.1, 2.1, 2.2.