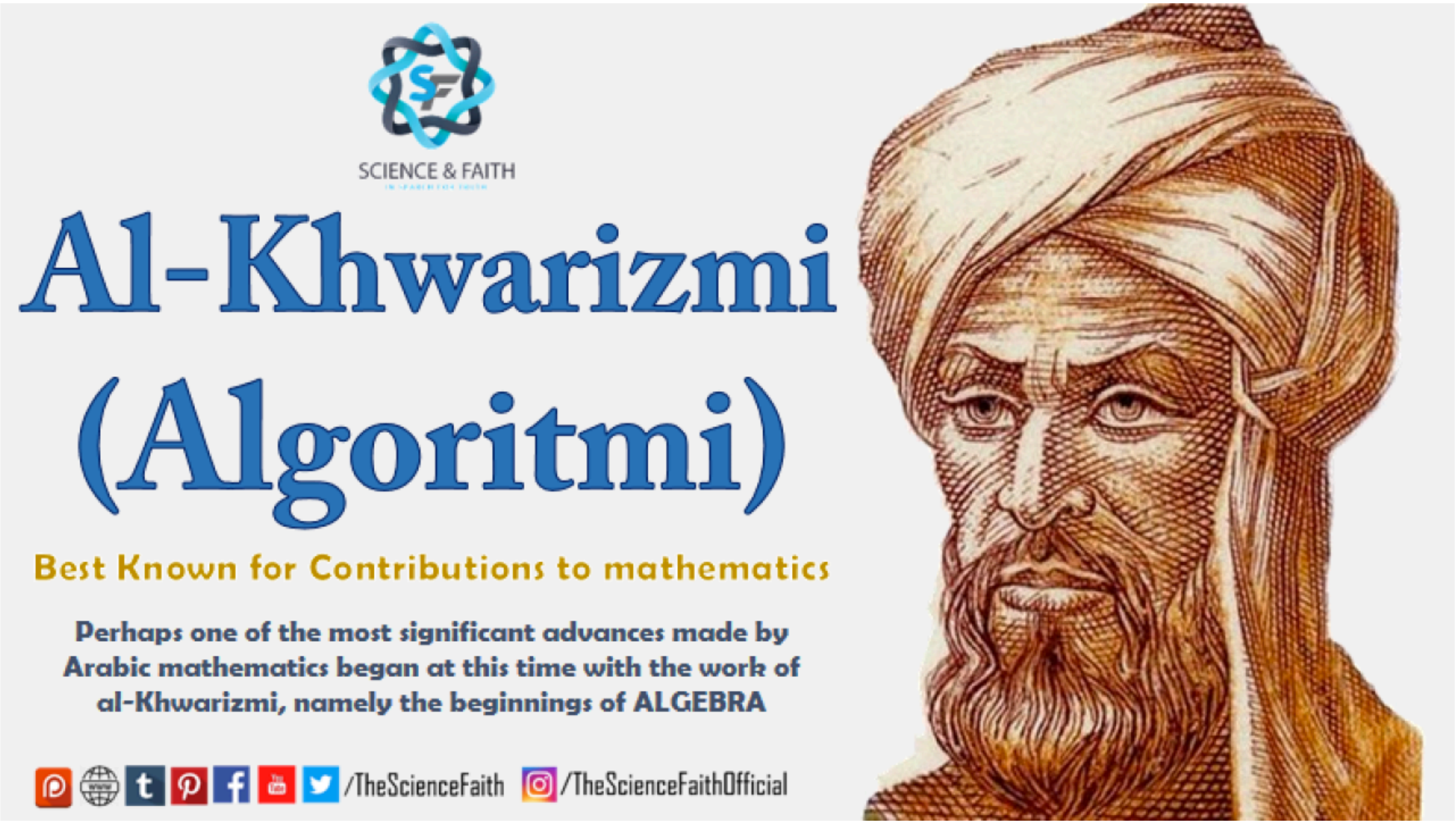
04 Complexity of Algorithms

CS201 Discrete Mathematics

Instructor: Shan Chen

Algorithms

- An algorithm is a finite sequence of precise instructions for performing a computation or for solving a problem.

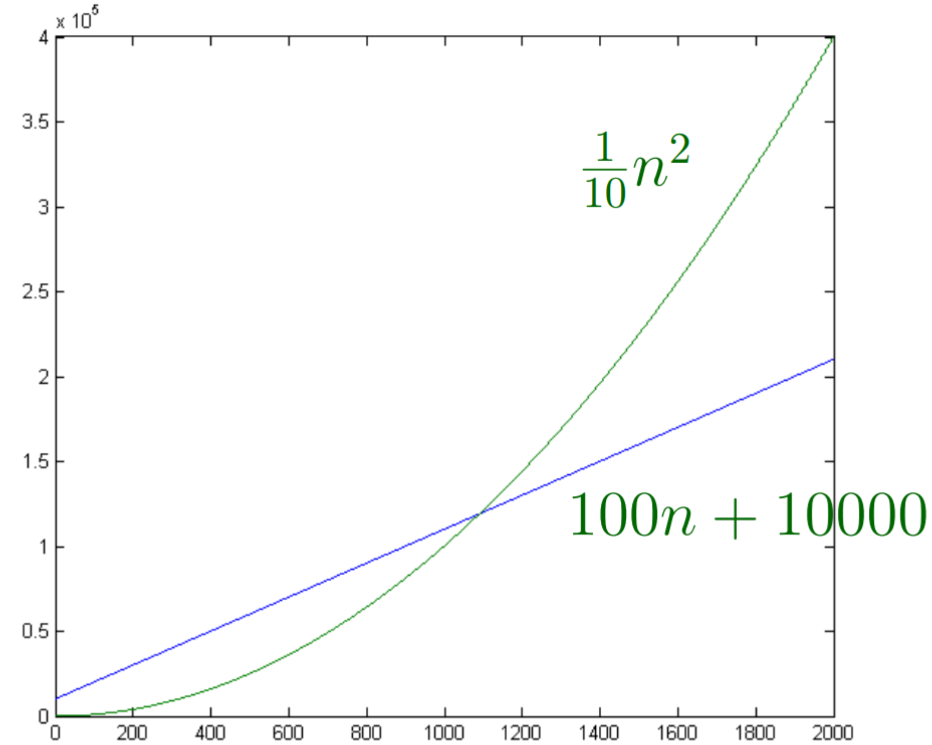


Al-Khwarizmi, Persian polymath

The Growth of Functions

Which Function is Larger?

- **Q:** Which function is "larger"? $n^2/10$ vs $100n + 10000$
- **A:** It depends on the value of n.
- In computer science, usually we are interested in what happens when the problem input size n gets big.
- Note that when n is "large enough", $n^2/10$ gets bigger than $100n + 10000$ and stays bigger for larger n.

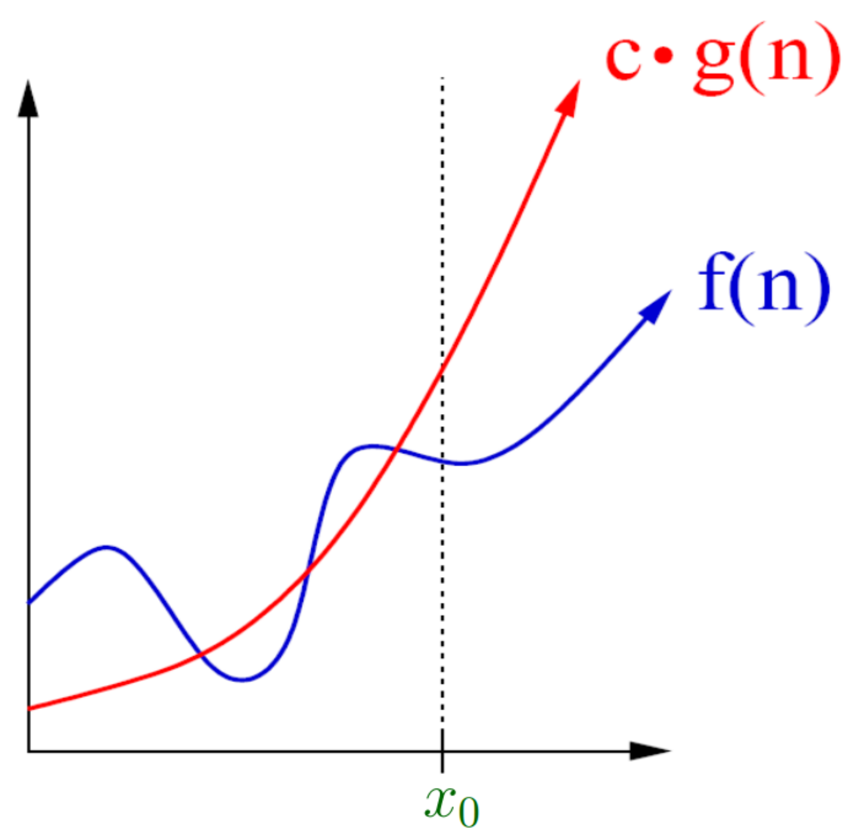


SUSTech

Big-O Notation

- **Definition:** Let f and g be functions from $\boldsymbol{Z}$ (or $\boldsymbol{R}$) to $\boldsymbol{R}$. We say that $f(x) = O(g(x))$ (read as *f(x) is big-oh of g(x)*), if there exist some positive constants c and x_0 such that
$$|f(x)| \leq c|g(x)|, \text{ whenever } x > x_0.$$

- Big-O gives an upper bound on the growth of a function. It tells us that a function grows at most as fast as the other function.



SUSTech

Big-O Notation

- Example: $100n + 10000 = O(n^2/10)$
  - Let $k = 2000$, we can verify that $\forall n > k,\ 100n + 10000 < n^2/10$
  - By definition, the opposite is not true, i.e., $n^2/10 \neq O(100n + 10000)$
- Some other $O(n^2)$ functions:
  - $4n^2$
  - $8n^2 + 2n - 3$
  - $n^2/5 + n^{1/2} - 10 \log n$
  - $n(n - 3)$

Big-O Estimates for Polynomials

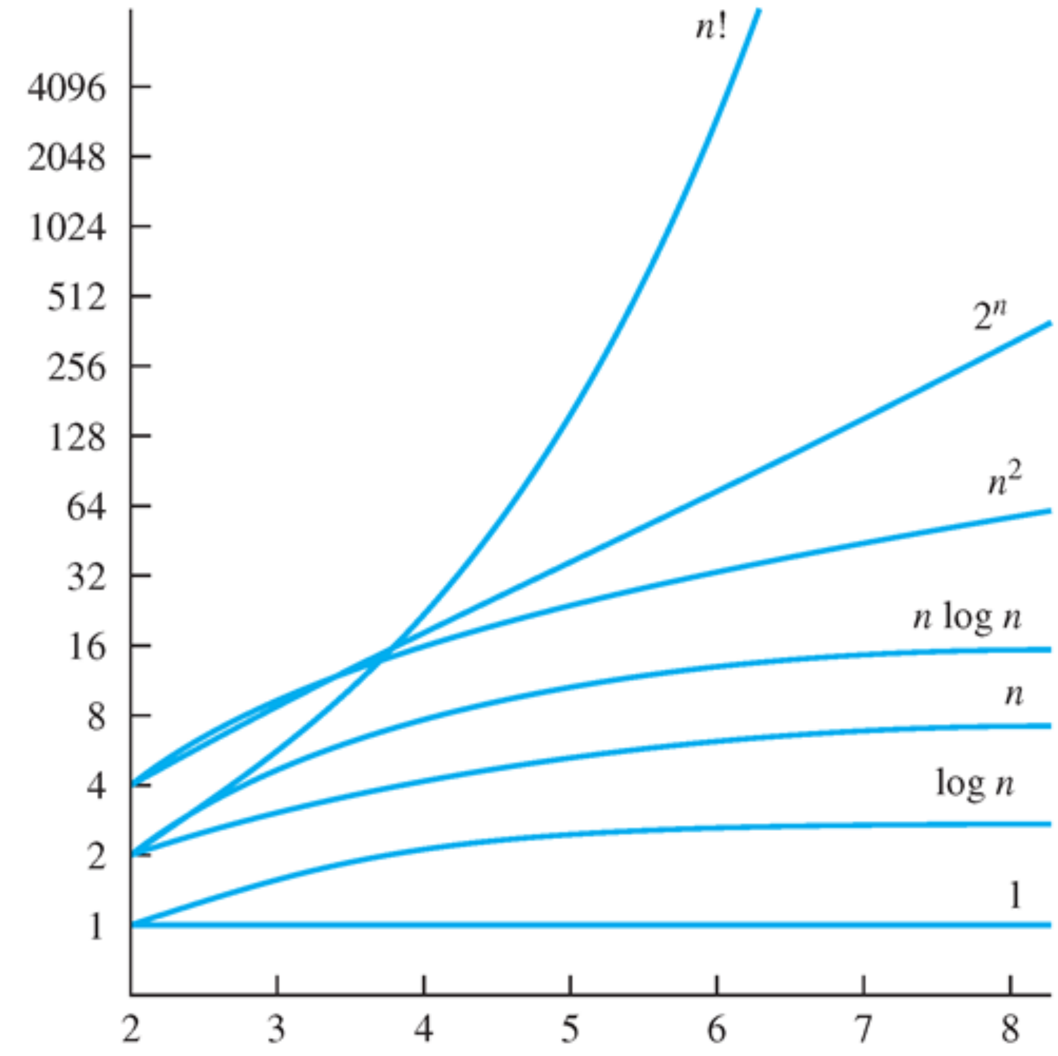
- **Theorem:** Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdot\cdot\cdot + a_1 x + a_0$, where a_0, a_1, …, a_n are real numbers. Then, $f(x) = O(x^n)$.
  - The leading term $a_n x^n$ of a polynomial dominates its growth.
- Proof:
  - Assuming $x > 1$, we have

$$
\begin{aligned}
|f(x)| &= |a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0| \\
&\leq |a_n| x^n + |a_{n-1}| x^{n-1} + \cdots + |a_1| x + |a_0| \\
&= x^n (|a_n| + |a_{n-1}|/x + \cdots + |a_1|/x^{n-1} + |a_0|/x^n) \\
&\leq x^n (|a_n| + |a_{n-1}| + \cdots + |a_1| + |a_0|)
\end{aligned}
$$

  - Choose $x_0 = 1$ and $c = |a_n| + |a_{n-1}| + \cdots + |a_1| + |a_0|$, then $|f(x)| \leq cx^n$ whenever $x > x_0$.

Some Big-O Estimates

- $1 + 2 + \cdots + n = O(n^2)$
- $n! = O(n^n)$
- $\log n! = O(n \log n)$
- $\log_a n = O(n)$ for $a > 0$
- $n^a = O(n^b)$ for $0 \leq a \leq b$
- $n^a = O(2^n)$



SUSTech

Combination of Functions

- **Theorem:** If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then

$$(f_1 + f_2)(x) = O(max(|g_1(x)|, |g_2(x)|))$$

- Proof:
  - By definition, there exist constants C_1, C_2, k_1, k_2 such that

    $|f_1(x)| \leq C_1|g_1(x)|$ *when* $x > k_1$

    $|f_2(x)| \leq C_2|g_2(x)|$ *when* $x > k_2$

  - Let $g(x) = max(|g_1(x)|, |g_2(x)|)$, when $x > max(k_1, k_2)$ we have

$$\begin{aligned} |(f_1 + f_2)(x)| &= |f_1(x) + f_2(x)| \leq |f_1(x)| + |f_2(x)| \\ &\leq C_1|g_1(x)| + C_2|g_2(x)| \leq C_1|g(x)| + C_2|g(x)| \\ &= (C_1 + C_2)|g(x)| \end{aligned}$$

  - The proof is concluded with $C = C_1 + C_2$ and $k = max(k_1, k_2)$.

Combination of Functions

- **Theorem:** If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then
  $$(f_1f_2)(x) = O(g_1g_2(x))$$
- Proof: *very similar to the previous theorem*
  - By definition, there exist constants C_1, C_2, k_1, k_2 such that
    $|f_1(x)| \le C_1|g_1(x)|$ when $x > k_1$
    $|f_2(x)| \le C_2|g_2(x)|$ when $x > k_2$
  - Let $g(x) = g_1g_2(x)$, when $x > max(k_1, k_2)$ we have
    $$\begin{aligned} |(f_1f_2)(x)| &= |f_1(x)f_2(x)| = |f_1(x)||f_2(x)| \\ &\le C_1|g_1(x)|C_2|g_2(x)| = C_1C_2|g_1(x)g_2(x)| \\ &= C_1C_2|g(x)| \end{aligned}$$
  - The proof is concluded with $C = C_1C_2$ and $k = max(k_1, k_2)$.

Exercise *(3 mins)*

- Order the following functions by order of growth:
  - $f_1(n) = (1.5)^n$
  - $f_2(n) = 8n^3 + 17n^2 + 111$
  - $f_3(n) = (\log n)^2$
  - $f_4(n) = 2^n$
  - $f_5(n) = \log(\log n)$
  - $f_6(n) = n^2(\log n)^3$
  - $f_7(n) = 2^n(n^2 + 1)$
  - $f_8(n) = 8n^3 + n(\log n)^2$
  - $f_9(n) = 100000$
  - $f_{10}(n) = n!$

Exercise (3 mins)

- Order the following functions by order of growth:
  - $f_1(n) = (1.5)^n$
  - $f_2(n) = 8n^3 + 17n^2 + 111$
  - $f_3(n) = (\log n)^2$
  - $f_4(n) = 2^n$
  - $f_5(n) = \log(\log n)$
  - $f_6(n) = n^2(\log n)^3$
  - $f_7(n) = 2^n(n^2 + 1)$
  - $f_8(n) = 8n^3 + n(\log n)^2$
  - $f_9(n) = 100000$
  - $f_{10}(n) = n!$
- Solution:
  - $f_9 < f_5 < f_3 < f_6 < f_8 < f_2 < f_1 < f_4 < f_7 < f_{10}$

Big-Ω Notation

- **Definition:** Let f and g be functions from $\boldsymbol{Z}$ (or $\boldsymbol{R}$) to $\boldsymbol{R}$. We say that $f(x) = \Omega(g(x))$ (read as *f(x) is big-omega of g(x)*), if there exist some positive constants c and x_0 such that
$$|f(x)| \geq c|g(x)|, \text{ whenever } x > x_0.$$

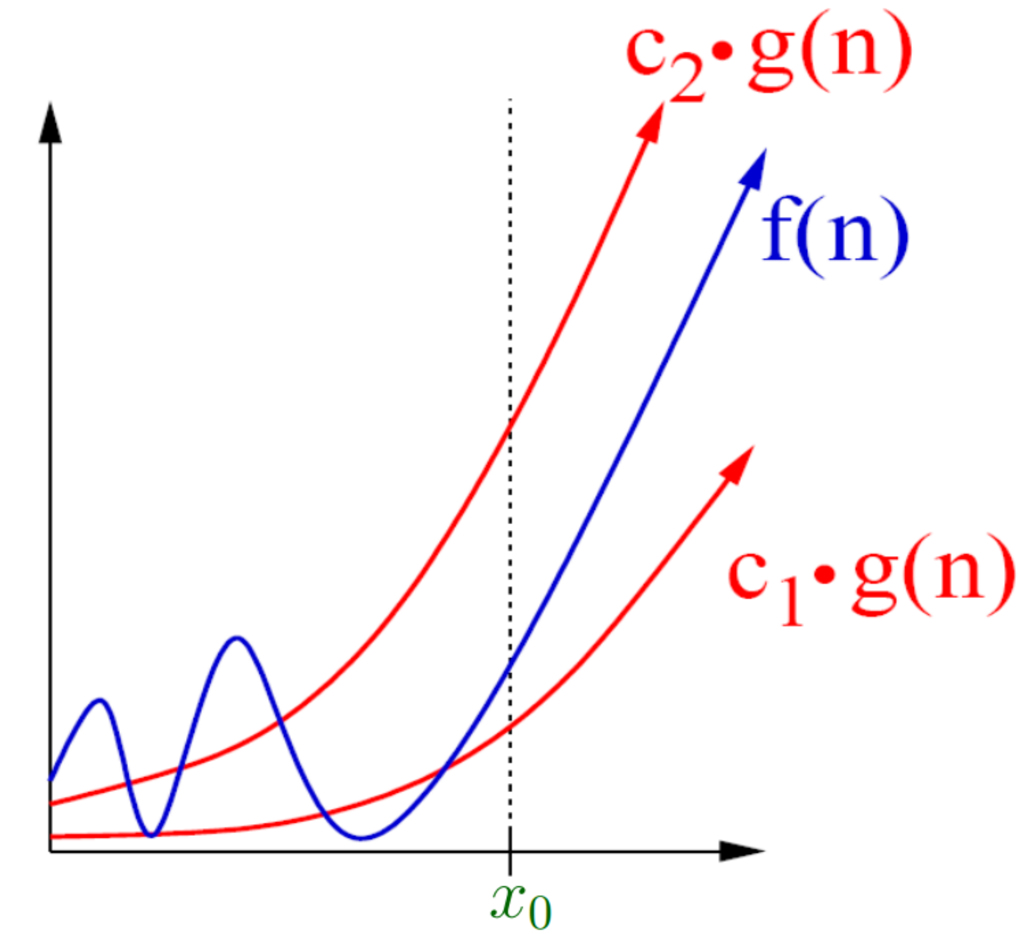
- Big-Ω gives a lower bound on the growth of a function. It tells us that a function grows at least as fast as the other function.

- Note: $f(x) = \Omega(g(x))$ if and only if $g(x) = O(f(x))$

Big-Θ Notation

- **Definition:** Let *f* and *g* be functions from ***Z*** (or ***R***) to ***R***. We say that *f(x)* = *Θ(g(x))* (read as *f(x) is big-theta of g(x)*), if they have the same order of growth: *f(x)* = *O(g(x))* and *f(x)* = *Ω(g(x))*.

- Note: *f(x)* = *Θ(g(x))* is equivalent to *g(x)* = *Θ(f(x))*



Exercise (3 mins)

- True or false?
  - $3n^2 + 4n = \Theta(n)$?
  - $3n^2 + 4n = \Theta(n^2)$?
  - $3n^2 + 4n = \Theta(n^3)$?
  - $n/5 + 10n \log n = \Theta(n^2)$?
  - $n^2/5 + 10n \log n = \Theta(n \log n)$?
  - $n^2/5 + 10n \log n = \Theta(n^2)$?

Exercise (3 mins)

- True or false?
  - $3n^2 + 4n = \Theta(n)$? No, but $\Omega(n)$
  - $3n^2 + 4n = \Theta(n^2)$? Yes
  - $3n^2 + 4n = \Theta(n^3)$? No, but $O(n^3)$
  - $n/5 + 10n \log n = \Theta(n^2)$? No, but $O(n^2)$
  - $n^2/5 + 10n \log n = \Theta(n \log n)$? No, but $\Omega(n \log n)$
  - $n^2/5 + 10n \log n = \Theta(n^2)$? Yes

Complexity of Algorithms

Computational Problems and Algorithms

- **Computational problem:** a task solved by a computer, which formally is a set of instances (i.e., problem input, with size n) together with a (perhaps empty) set of solutions (problem output) for every instance.
    - An instance is just a specific problem input, not the problem itself.

- **Algorithm:** a finite sequence of precise instructions for performing a computation or for solving a problem.

- We say an algorithm solves the problem if it halts (ends) with the correct output for every input instance.

SUSTech

Computational Problems and Algorithms

- **Computational problem:** a task solved by a computer.
- **Algorithm:** a finite sequence of precise instructions for performing a computation or for solving a problem.
- We say an algorithm solves the problem if it halts with the correct output for every input instance
- Example: algorithm for calculating the sum of $a_1, a_2, \ldots, a_n$
  - Step 1: set $S = 0$
  - Step 2: for $i = 1$ to n, $S := S + a_i$ (i.e., assign S the value $S + a_i$)
  - Step 3: output S

  * *problem instance example: < 8, 3, 6, 7, 1, 2, 9 > (here $n = 7$)*

SUSTech

Time and Space Complexity

- **Time complexity:** the number of machine operations (addition, multiplication, comparison, assignment, etc.) in an algorithm

- **Space complexity:** the amount of memory in an algorithm

- Example: algorithm for calculating the sum of $a_1, a_2, \ldots, a_n$
  - Step 1: set $S = 0$
  - Step 2: for $i = 1$ to n, $S := S + a_i$ (i.e., assign S the value $S + a_i$)
  - Step 3: output S
  - **time complexity:** $O(n)$ ** usually we ignore operations on iterator i*

    Step 2 takes n operations (in-place additions). Step 1 and 3 each take *1* operation. Altogether this algorithm takes $n + 2$ operations.
  - **space complexity**: $O(n)$

    The input numbers take $O(n)$ memory and S, i take $O(1)$ memory.

Example: Horner's Method

- Example: consider the evaluation of $f(x) = 1 + 2x + 3x^2 + 4x^3$
  - **direct computation:** 3 additions and 6 multiplications
  - **better solution:** evaluate $f(x) = 1 + x(2 + x(3 + 4x))$ instead, which takes 3 additions and 3 multiplications

- **Polynomial evaluation:** $f(x) = a_0 + a_1x + \cdots + a_nx^n$

- **Horner's method:** $f(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n) \cdots))$
  - Step 1: set $S = a_n$
  - Step 2: for $i = 1$ to n, $S := a_{n-i} + xS$
  - Step 3: output S
  - **time complexity:** $O(n)$

    Step 1 and 3 each take *one* operation. Step 2 takes $3n$ operations: n multiplications, n additions, n assignments.

Another Example

- Determine the time complexity of the following algorithm:

$$
\begin{array}{l}
\text{for } i := 1 \text{ to } n \\
\quad \text{for } j := 1 \text{ to } n \\
\quad\quad a := 2 * n + i * j; \\
\quad \text{end for} \\
\text{end for}
\end{array}
$$

- Computing the value of *a* in each iteration takes *4* operations (two multiplications, one addition and one assignment). There are n^2 iterations in two loops. So it takes $n^2 \times 4 = 4n^2$ operations. The time complexity of this algorithm is $O(n^2)$.
    - Note that we can compute $2 * n$ only once but still $O(n^2)$ complexity.

Exercise (3 mins)

- Determine the time complexity of the following algorithm:

$$
\begin{array}{l}
S := 0 \\
\text{for } i := 1 \text{ to } n \\
\quad \text{for } j := 1 \text{ to } i \\
\quad\quad S := S + i * j; \\
\quad \text{end for} \\
\text{end for}
\end{array}
$$

Exercise *(3 mins)*

- Determine the time complexity of the following algorithm:

$$
\begin{array}{l}
S := 0 \\
\text{for } i := 1 \text{ to } n \\
\quad \text{for } j := 1 \text{ to } i \\
\quad\quad S := S + i * j; \\
\quad \text{end for} \\
\text{end for}
\end{array}
$$

- The first S assignment takes *1* operation. Computing the value of S in each iteration takes 2 operations (one multiplication and one in-place addition). There are $1 + 2 + \ldots + n = n(n + 1)/2$ iterations in two loops, so it takes $1 + n(n + 1)/2 \times 2 = n^2 + n + 1$ operations. The time complexity of this algorithm is $O(n^2)$.

SUSTech

Types of Complexity Analysis

- Example: (Insertion Sort)

Input: $A[1 \ldots n]$ is an array of numbers

for $j := 2$ to n

  $key = A[j]$;

  $i = j - 1$;

  while $i \geq 1$ and $A[i] > key$ do

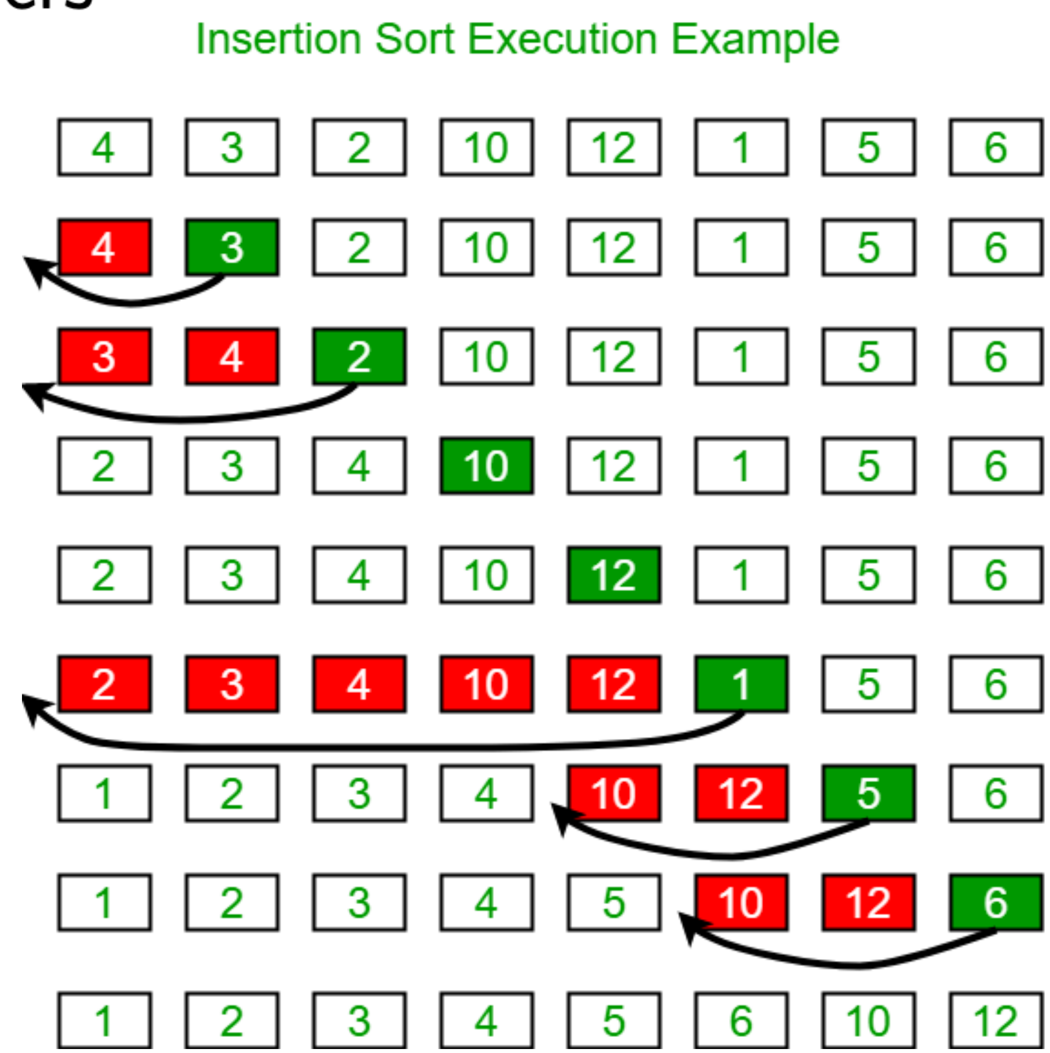
    $A[i + 1] = A[i]$;

    $i - -$;

  end while

  $A[i + 1] = key$;

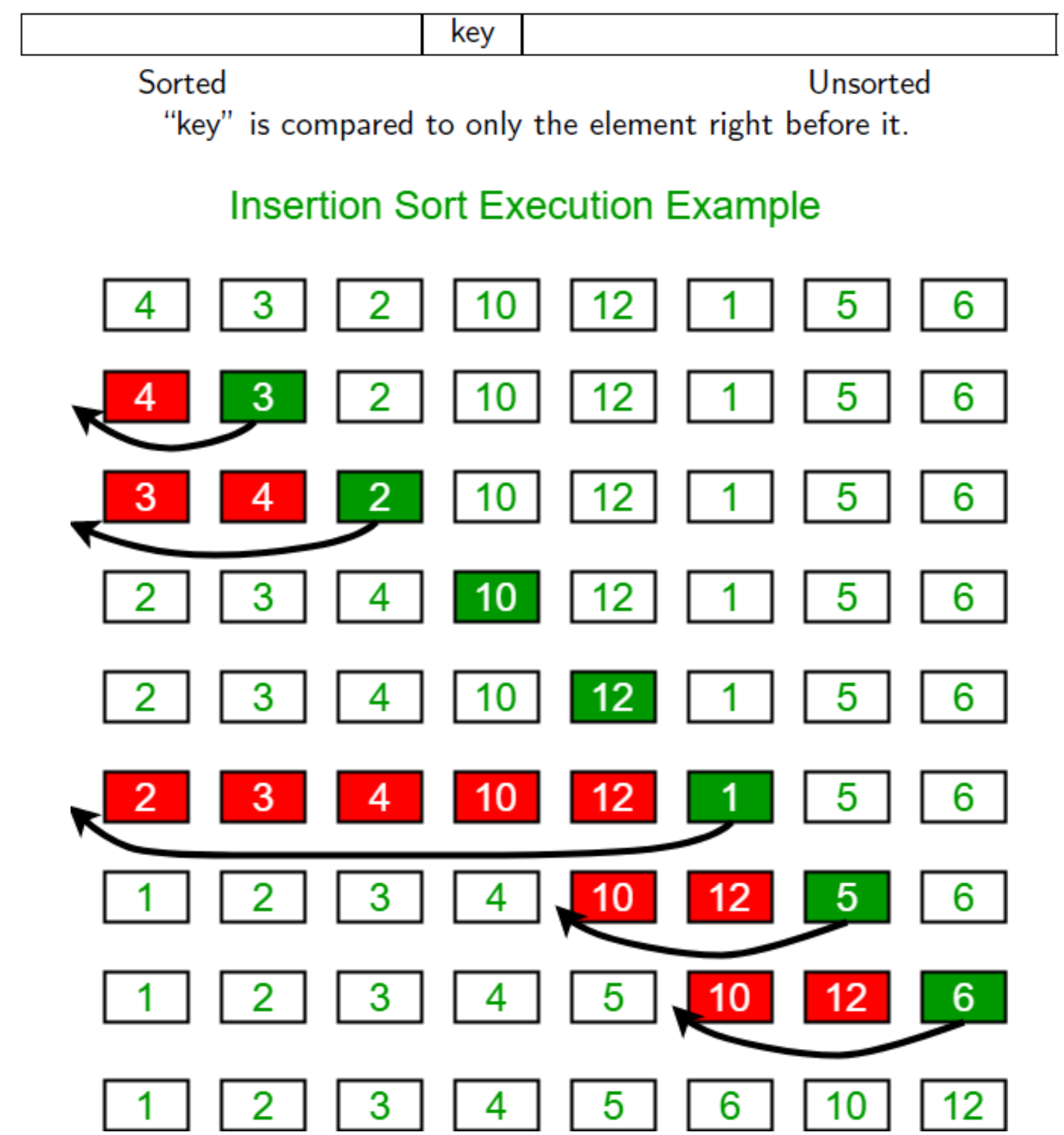
end for



Complexity Analysis: Type I

- **Best-case complexity:**
  - constraints on the input rather than size
  - resulting in the fastest possible running time for the given size.
- Example: (Insertion Sort)
  - $A[1] \leq A[2] \leq A[3] \leq \cdots \leq A[n]$
  - **time complexity:** *Θ(n)*

    n - 1 comparisons



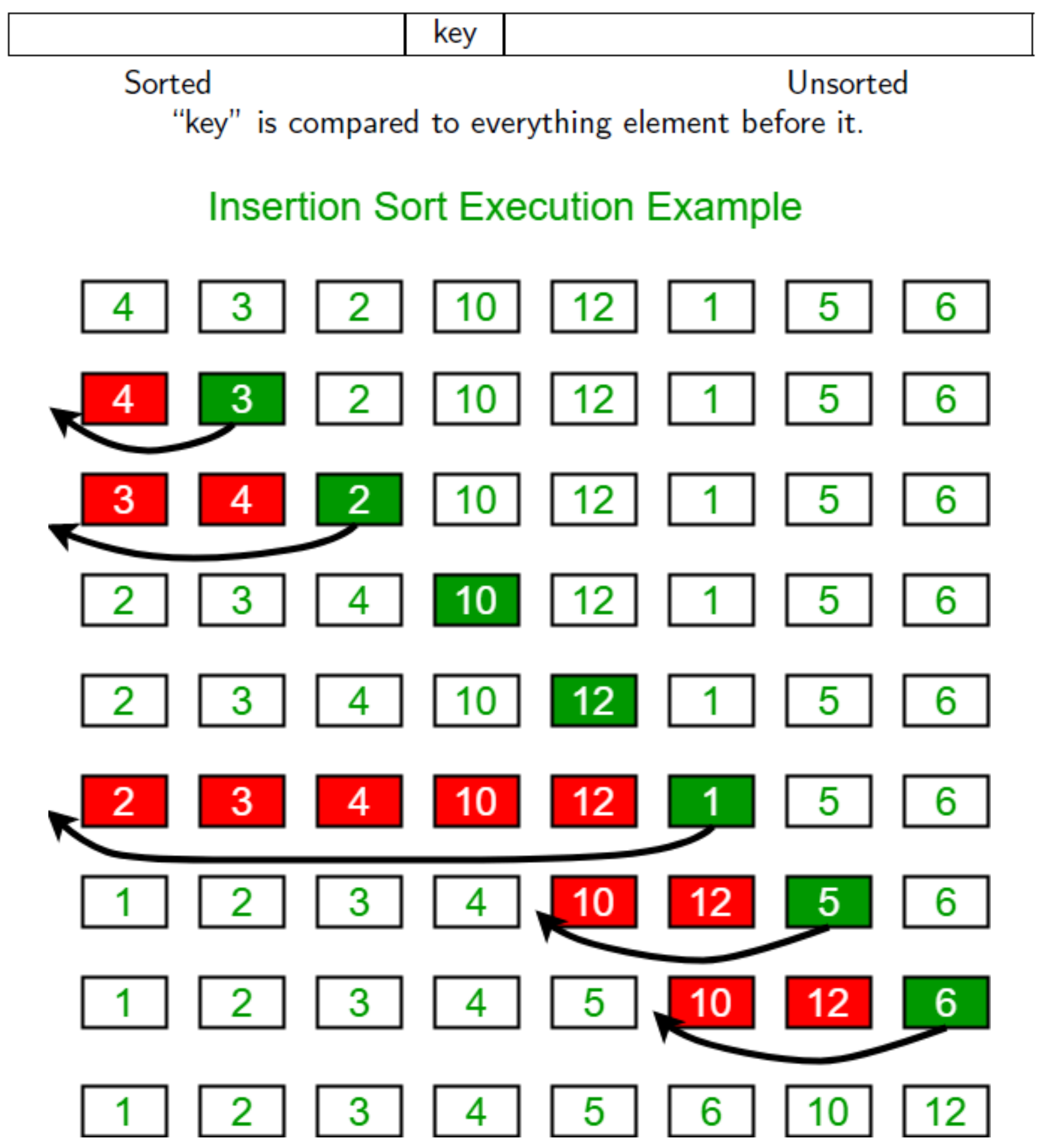
SUSTech

Complexity Analysis: Type II

- **Worst-case complexity:**
  - constraints on the input rather than size
  - resulting in the slowest possible running time for the given size.
- Example: (Insertion Sort)
  - A[1] ≥ A[2] ≥ A[3] ≥ · · · ≥ A[n]
  - **time complexity:** *Θ(n²)*

$$\sum_{j=2}^{n} j-1 = \frac{n(n-1)}{2}$$

comparisons & swaps

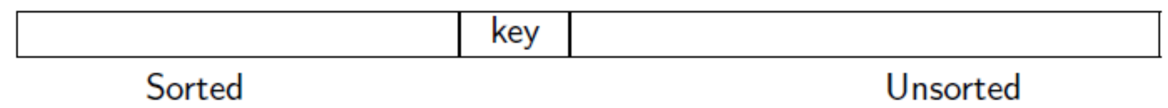


Complexity Analysis: Type III

- **Average-case complexity:**
  - constraints on the input rather than size
  - average running time over all possible inputs for the given size (usually involving probability distribution on input instances)
- Example: (Insertion Sort)
  - if *n!* instances are equally likely
  - **time complexity:** $\Theta(n^2)$

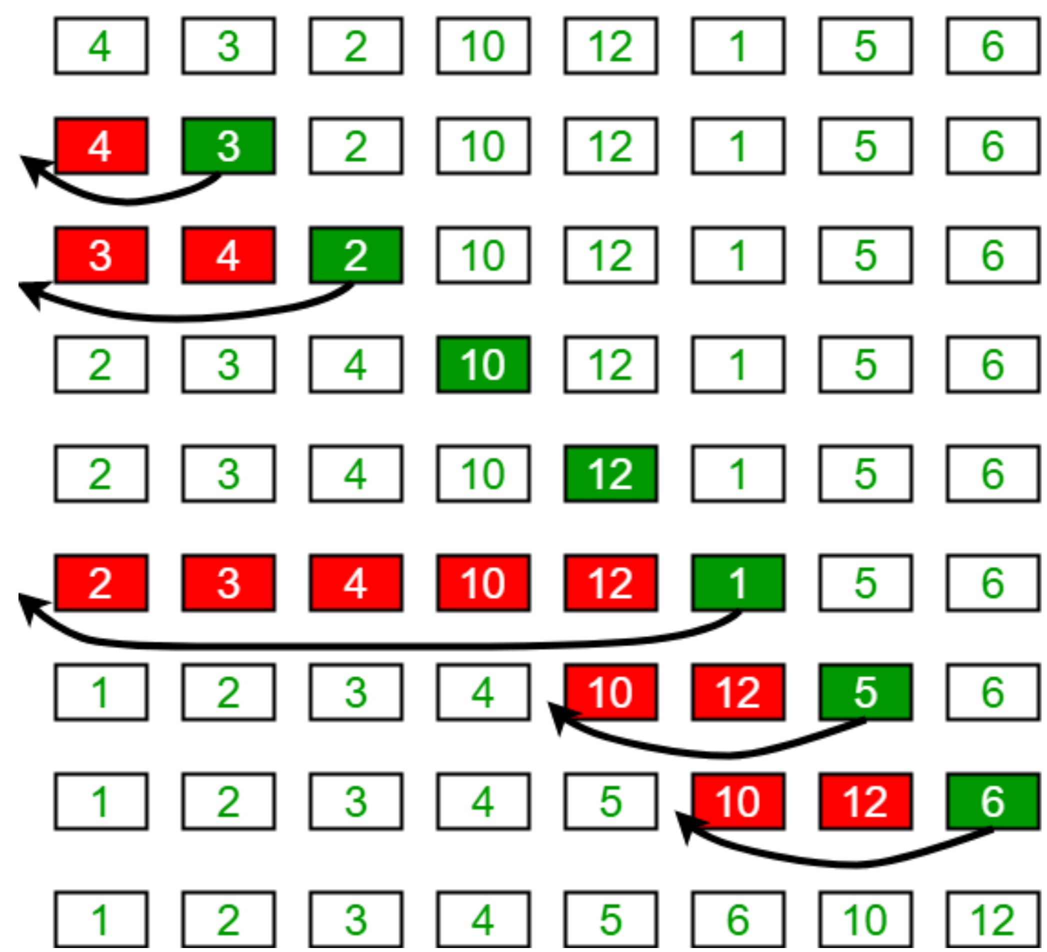
$$\sum_{j=2}^{n} \frac{j-1}{2} = \frac{n(n-1)}{4}$$

comparisons & swaps



On average, "key" is compared to half of the elements before it.

Insertion Sort Execution Example



SUSTech

Some Thoughts on Algorithm Design

- Algorithm design is mainly about designing algorithms that have small Big-O running time.
- Being able to design good algorithms lets you identify the hard parts of your problem and handle them effectively.
- Too often, programmers try to solve problems using brute force techniques and end up with slow and complicated code!
- A few hours of abstract thought devoted to algorithm design could have speeded up and simplified the solution substantially!

Complexity of Problems

Dealing with Hard Problems

- What would you do if you cannot find an efficient algorithm for a given problem?

Blame yourself

Prove that no such algorithm exists

SUSTech

Dealing with Hard Problems

- Showing that a problem has efficient algorithms is relatively easy:
  - All we have to do is to demonstrate an algorithm.
- Proving that no efficient algorithm exists for a particular problem is difficult:
  - How can we prove the non-existence of something?
- We will now learn about ***NP***-complete problems, which provide us with a way to approach this question.

Introduction to *NP*-Complete

- ***NP*-complete problems:** a very large class of problems (> 3000 are known) which is not known to have any "efficient" solutions.
- It is known that if any one of the ***NP***-complete problems has an efficient solution then all of the ***NP***-complete problems have efficient solutions.
- Researchers have spent innumerable man-years trying to find efficient solutions to ***NP***-complete problems but failed.
- So, ***NP***-complete problems are very likely to be hard.
- What we can do: prove that a hard problem is ***NP***-complete.
  - This shows no one can find an efficient solution so far.
- Next, we show how to define such complexity classes formally.

SUSTech

Example Problem: COMPOSITE

- **COMPOSITE:** given a positive integer n, are there integers $d, k \geq 2$ such that $n = dk$?

- The naive algorithm for determining whether n is composite is to enumerate d from 2 to $n - 1$ to see if any of them divides n.
  - This takes $\Theta(n)$ division operations, which might look like linear time and very efficient. However, it is problematic to treat the value of n as the input size of the algorithm, because integer n is usually processed as a binary string of length $\Theta(\log_2 n)$ rather than $\Theta(n)$. An efficient algorithm should have time complexity "close" to its input size $\Theta(\log_2 n)$ rather than the input value n.

    e.g., $n \times n$ requires only $O((\log_2 n)^2)$ bit operations (show later)
  - Therefore, the input size of COMPOSITE is $L = \log_2 n$. Then, the time complexity is $\Theta(n) = \Theta(2^L)$, i.e., exponential in the input size L and hence very impractical. (Note that integer division n/d also takes $O((\log_2 n)^2)$ bit operations, which we ignore for simplicity.)

- **Takeaway:** we should use the input size to measure complexity.

SUSTech

The Input Size of Problems

- Complexity of a problem is measured in terms of its input size.
  - The input size of a problem is the number of bits needed to encode the input of the problem.
- The optimal input size, determined by an optimal encoding method, is hard to compute in most cases.
- For most problems, it is sufficient to choose some natural, and (usually) simple, encoding method and use its encoded input size.
- Example 1: COMPOSITE
  - What is the input size of this problem?

    Any integer $n \geq 1$ can be represented as a binary string $a_0 a_1 \cdots a_L$ of length $\lceil \log_2 (n + 1) \rceil$. Therefore, a natural measure of the input size is $\lceil \log_2 (n + 1) \rceil$ (or $\Theta(\log_2 n)$ for simplicity)

The Input Size of Problems

- Complexity of a problem is measured in terms of its input size.
  - The input size of a problem is the number of bits needed to encode the input of the problem.

- The optimal input size, determined by an optimal encoding method, is hard to compute in most cases.

- For most problems, it is sufficient to choose some natural, and (usually) simple, encoding method and use its encoded input size.

- Example 2: Sort n integers $a_1, \ldots, a_n$
  - What is the input size of this problem?

    Fixed-length encoding: all input numbers share the same length. We write every input integer a_i as a binary string of the same length $m = \lceil \log_2 \max(|a_i| + 1) \rceil + 1$ (one extra bit for the +/- sign).

    This natural encoding gives an input size nm.

SUSTech

Decision and Optimization Problems

- **Decision problem:** a problem that has a yes or no answer.
  - E.g., "Given $n > 0$, is integer m such that $m^m < n$?"
- **Optimization problem:** a problem that asks for some answer that maximizes or minimizes a particular objective function.
  - E.g., "Given $n > 0$, what is the largest integer m such that $m^m < n$?"
- Given an algorithm for solving the optimization problem, solving the corresponding decision problem is usually trivial.
  - Contrapositive: if we prove that a given decision problem is hard to solve efficiently, then the corresponding optimization problem must be (at least as) hard.
- The other direction (decision → optimization) also often works.
  - E.g., use binary search to find m in the above examples.

Complexity Classes

- **Computational complexity theory** is a field that deals with:
  - classification of certain "decision problems" into several classes:
    the class of "easy" problems
    the class of "hard" problems
    the class of "hardest" problems
  - relations among the above classes
  - properties of problems in the above classes
- How to classify decision problems?
  - use polynomial-time algorithms (often called efficient algorithms)

Polynomial-Time Algorithms

- **Polynomial-time algorithm:** an algorithm that runs in time $O(n^c)$, where $c > 0$ is a constant number independent of n, and n is the input size of the problem that the algorithm solves.
  - E.g., popular sorting algorithms are polynomial-time algorithms.
- Our expectations:
  - When the input size of the algorithm is n^a (for any constant $a > 0$), the algorithm should still be polynomial-time.
  - Also, an algorithm that is composed by several polynomial-time algorithms should still be polynomial-time.
- The above somehow shows why people choose polynomial-time to define efficient algorithms, because the common operations (e.g., addition, subtraction, multiplication, composition, etc.) are closed for polynomials.

SUSTech

Non-Polynomial-Time Algorithms

- **Non-polynomial-time algorithm:** an algorithm of which the running time is not $O(n^c)$ for any constant $c > 0$.
  - E.g., naive algorithm for solving the composite number problem
- Non-polynomial-time algorithms are usually impractical.
  - E.g., exponential-time 2^n for $n = 100$ takes billions of years!!!
- **Caveat:** even polynomial-time algorithms could be impractical.
  - E.g., a $\Theta(n^{20})$ algorithm may not be very practicalbb for $n = 100$.

Tractable Problems and Class *P*

- **Tractable problem:** a problem that is solvable in polynomial time (or the problem is in polynomial time). That is, there exists a polynomial-time algorithm that solves the problem.
- **Class *P*** consists of all decision problems that are solvable in polynomial time. That is, there exists a polynomial-time algorithm that decides if any given input is a yes-input or a no-input.
  - E.g., PRIMES (determining whether a number is prime) is in ***P***.
- How to prove that a decision problem is in ***P***?
  - find a polynomial-time algorithm (relatively easy)
- How to prove that a decision problem is not in ***P***?
  - prove that there is no polynomial-time algorithm for solving this problem (much much harder)

Certificates and Class *NP*

- A decision problem is usually formulated as: "Is there an object satisfying some conditions?"

- A proof/certificate/witness for a yes-input is a specific object that is used to verify/prove/show that this input is indeed a yes-input.
    - E.g., the COMPOSITE problem can be formulated as: "Is there an integer d $(1 < d < n)$ such that d divides n?". So, a certificate for a composite number n (i.e., n is a yes-input of COMPOSITE) can be one of such integer factors d.

- **Class *NP*** (nondeterministic polynomial-time) consists of all decision problems that: there exists a polynomial-time algorithm V such that, for each yes-input, there is a proof/certificate/witness with which V can verify the input is indeed a yes-input.
    - E.g., COMPOSITE is in ***NP*** because the certificate can be verified in polynomial time (in the input size): the input size is $\Theta(\log_2 n)$ and checking if d divides n takes $O((\log_2 n)^2)$ bit operations.

P = *NP*?

- Whether ***P*** = ***NP*** is one of the most important problems in CS.
- It is not hard to see that ***P*** ⊆ ***NP***. * *why?*
- Intuitively, ***NP*** ⊆ ***P*** is doubtful.
  - Just being able to verify a certificate in polynomial time does not necessarily mean we can tell whether an input is a yes-input or a no-input in polynomial time, e.g., certificates may be hard to find.
  - So far, we are still far from solving it and do not know the answer. However, the search for such a solution has provided us with deep insights into what distinguishes “easy” problems from “hard” ones.

***NP*-Complete and *NP*-Hard**

- ***NP*-complete:** consists of the hardest problems in ***NP***.
  - ***NP***-complete problems are reducible to each other, i.e., they are equivalently hard

    If solving problem A can be transformed into solving problem B, we say A reduces to B. This also means B is at least as hard as A.

- ***NP*-hard:** consists of problems at least as hard as ***NP***-complete.
  - Some ***NP***-hard problems may not belong to ***NP***.

05 Number Theory and Cryptography

To be continued…

Announcements

- Please submit your Undergraduate Students Declaration Form with your handwritten signature in Assignment 0 if you have not yet done so.
- Assignment 2 was already released and is due on Oct 23:
  - 100 points maximum but 110 in total
  - DO NOT CHEAT!