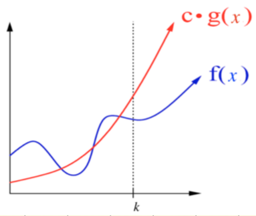
The Growth of Functions

Big-O notation

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \le C|g(x)|,$$

whenever $x > k$. [This is read as "$f(x)$ is big-oh of $g(x)$."]



Big-O Estimates for Some Functions

$1 + 2 + \cdots + n = O(n^2)$

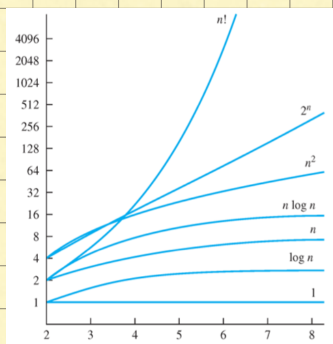
$n! = O(n^n)$

$\log n! = O(n \log n)$

$\log_a n = O(n)$ for an integer $a \ge 2$

$n^a = O(n^b)$ for integers $a \le b$

$n^a = O(2^n)$ for an integer a



Combinations of Functions

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 $|f_1(x)| \le C_1|g_1(x)|$ when $x > k_1$ and $|f_2(x)| \le C_2|g_2(x)|$ when $x > k_2$. Then

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

where $g(x) = \max(|g_1(x)|, |g_2(x)|)$ and $C = C_1 + C_2$.

$k = \max\{k_1, k_2\}$.

If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ then $(f_1 f_2)(x) = O(g_1(x) g_2(x))$.

Proof:

When k $> \max(k_1, k_2)$,

$$\begin{aligned}
|(f_1 f_2)(x)| &= |f_1(x)||f_2(x)| \\
&\le C_1|g_1(x)|C_2|g_2(x)| \\
&\le C_1 C_2|(g_1 g_2)(x)| \\
&\le C|(g_1 g_2)(x)|,
\end{aligned}$$

where $C = C_1 C_2$.

Theorem: Let $f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$, where $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \ne 0$. Then $f(x)$ is of order x^n.

- $f(x) = O(x^n)$
- $f(x) = \Omega(x^n)$

Big-Omega Notation

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$ if there are positive constants C and k such that

$$|f(x)| \ge C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-Omega of $g(x)$."]

Big-O gives an upper bound on the growth of a function, while Big-Ω gives a lower bound.

Big-Ω tells us that a function grows at least as fast as another.

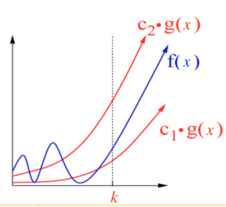
Note: $f(x)$ is $\Omega(g(x))$ if and only if $g(x)$ is $O(f(x))$.

Big-Theta Notation

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Theta(g(x))$ if

- $f(x)$ is $O(g(x))$ and
- $f(x)$ is $\Omega(g(x))$.

When $f(x)$ is $\Theta(g(x))$, we say that $f(x)$ is big-Theta of $g(x)$, that $f(x)$ is of order $g(x)$, and that $f(x)$ and $g(x)$ are of the same order.



Algorithms

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

A computational problem is a specification of the desired input-output relationship.

Example (Computational Problem and Algorithm):

- Computational Problem: Input n numbers $a_1, a_2, \ldots, a_n$; Output the sum of the n numbers.
- Algorithm: the following procedures

  Step 1: set $S = 0$

  Step 2: for $i = 1$ to n, replace S by $S + a_i$

  Step 3: output S

Instance & correct algorithm.

An instance of a problem is a realization of all the inputs needed to compute a solution to the problem.

Example: $8, 3, 6, 7, 1, 2, 9$

A correct algorithm halts with the correct output for every input instance. We can then say that the algorithm solves the problem.

Time and Space Complexity

- Time complexity: The number of machine operations (addition, multiplication, comparison, replacement, etc) needed in an algorithm.
- Space complexity: the amount of memory needed.

Horner's Algorithm and its Complexity

Horner's algorithm for computing

$f(x) = a_0 + a_1x + \ldots + a_{n-1}x^{n-1} + a_nx^n = a_0 + x(a_1 + \ldots + x(a_{n-1} + a_nx))$

at a particular x:

Step 1: set $S = a_n$

Step 2: for $i = 1$ to n, replace S by $a_{n-i} + Sx$

Step 3: output S

The number of operations needed in this algorithm is $1 + 3n + 1 = 3n + 2$. So the time complexity of this algorithm is $O(n)$.

Note: Operations: addition, multiplication, comparison, replacement, etc.

Three Cases of Analysis:

Best-case:

Best-Case Complexity: The smallest number of operations needed to solve the given problem using this algorithm on input of specified size.

Example: (Insertion Sort)

$$A[1] \leq A[2] \leq A[3] \leq \cdots \leq A[n]$$

The number of comparisons needed is

$$\underbrace{1 + 1 + 1 + \cdots + 1}_{n-1} = n - 1 = \Theta(n)$$

Sorted	key	Unsorted

"key" is compared to only the element right before it.

Worst-Case

Worst-Case Complexity: The largest number of operations needed to solve the given problem using this algorithm on input of specified size.

Example: (Insertion Sort)

$$A[1] \geq A[2] \geq A[3] \geq \cdots \geq A[n]$$

The number of comparisons needed is

$$1 + 2 + 3 + \cdots + (n-1) = \frac{n(n-1)}{2} = \Theta(n^2)$$

Sorted	key	Unsorted

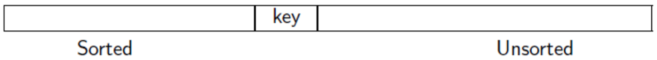
"key" is compared to everything element before it.

Average-Case

Average-Case Complexity: The average number of operations used to solve the problem over all possible inputs of a given size is found in this type of analysis.

Example: (Insertion Sort)

$\Theta(n^2)$ assuming that each of the $n!$ instances are equally likely



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

- For a particular instance, compute the number of comparisons
- Since we assume equal probability, take the average

Average-case complexity is usually difficult to compute.

Algorithm Design

Algorithm Design is mainly about designing algorithms that have small Big-O running time.

Being able to do good algorithm design lets you identify the hard parts of your problem and deal with them effectively.

Too often, programmers try to solve problems using brute force techniques and end up with slow complicated code!

- The most straightforward manner based on the statement of the problem and the definitions of terms

A few hours of abstract thought devoted to algorithm design could speed up the solution substantially and simplified it!

Showing that a problem has an efficient algorithm is, relatively easy:

- Design such an algorithm.

Proving that no efficient algorithm exists for a particular problem is difficult:

How can we prove the non-existence of something?

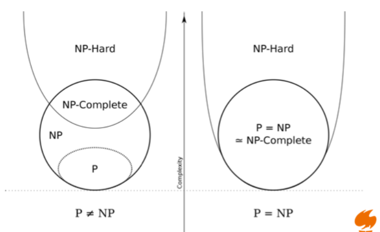
We will now learn about NP-Complete problems, which provides us with a way to approach this question.

NP-Complete

P: Problems that are solvable using an algorithm with polynomial worst-case complexity

NP: Problems for which a solution can be checked in polynomial time.

NP-Complete: If any of these problems can be solved by a polynomial worst-case time algorithm, then all problems in the class NP can be solved by polynomial worst-case time algorithms.



Researchers have spent many years trying to find efficient solutions to these problems but failed.

NP-Complete and NP-Hard problems are very likely to be hard.

Thus, to proving that no efficient algorithm exists for a particular problem?

Prove that your problem is NP-Complete or even NP-Hard:

- Show that your problem can be reduced to a typical (well-known) NP-Complete or NP-Hard problem.

Encoding the Inputs of Problems

Complexity of a problem is measure with respect to the size of input:

- E.g., for insertion sort, $\Theta(n^2)$ is the average-case complexity, where n is the length of the array.

In order to formally discuss how hard a problem is, we need to be much more formal than before about the input size of a problem.

The Input Size of Problems

The input size of a problem might be defined in a number of ways.

Now, we consider the following definition:
Definition: The input size of a problem is the minimum number of bits (i.e., $\{0, 1\}$) needed to encode the input of the problem.

The exact input size s, 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 and use the size s of this encoding.

- E.g., 5 can be encoded as 101.

Complexity in terms of Input Size

Example (Composite): The naive algorithm for determining whether n is composite compares n with the first $n-1$ numbers to see if any of them divides n.

This makes $\Theta(n)$ comparisons, so it might seem linear and very efficient.

But, the input size of this problem is $\log_2 n$ instead of n. The number of comparisons performed is actually $\Theta(n)$, which can be represented as $\Theta(2^{(\log_2 n)})$. It is exponential with respect to the input size.

Functions of the Same Type

Definition: Two positive functions $f(n)$ and $g(n)$ are of the same type if

$$c_1 g(n^{a_1})^{b_1} \le f(n) \le c_2 g(n^{a_2})^{b_2}$$

for all large n, where a_1, b_1, c_1, a_2, b_2, c_2 are some positive constants.

Example:

- All polynomials are of the same type
- Polynomials and exponentials are of different types.

Decision Problems and Optimization Problem

Definition: A decision problem is a question that has two possible answers: yes and no.

Definition: An optimization problem requires an answer that is an optimal configuration.

- Decision variables
- Maximize or minimize certain objective subject to some constraints

An optimization problem usually has a corresponding decision problem.

Examples:
Knapsack vs. Decision Knapsack (DKnapsack)

Given a subroutine for solving the optimization problem, solving the corresponding decision problem is usually trivial.

- First, solve the optimization problem
- Then, check the decision problem.

Thus, if we prove that a given decision problem is hard to solve efficiently, then it is obvious that the optimization problem must be (at least as) hard.

Knapsack vs DKnapsack

We have a knapsack of capacity W (a positive integer) and N objects with weights $w_1, \ldots, w_N$ and values $v_1, \ldots, v_N$, where v_n and w_n are positive integers.

Optimization problem (Knapsack):

- Decision variable $x_n \in \{0, 1\}$: $x_n = 1$, object x is placed in the knapsack; $x_n = 0$, otherwise
- Maximize $\sum_{n=\{1,\ldots,N\}} x_n v_n$, subject to constraint $\sum_{n=\{1,\ldots,N\}} x_n w_n \leq W$.

Decision problem (DKnapsack): Given V, is there a subset of the objects that fits in the knapsack and has total value at least V?

The optimization problem is at least as hard as the decision problem.

Complexity Classes

Theory of Complexity deals with

1. the classification of certain "decision problems" into several classes:
   - the class of "easy" problems
   - the class of "hard" problems
   - the class of "hardest" problems
2. relations among the three classes
3. properties of problems in the three classes

Question: How to classify decision problems?

Answer: Use polynomial-time algorithms.

P problem, NP problem, ...

Polynomial-Time Algorithm

Definition: An algorithm is polynomial-time if its running time is $O(n^k)$, where k is a constant independent of n, and n is the input size of the problem that the algorithm solves.

Whether we use n or n^a (for a fixed $a > 0$) as the input size, it will not affect the conclusion of whether an algorithm is polynomial-time.

Example:
The standard multiplication algorithm has time $O(m_1 m_2)$, where m_1 and m_2 denote the number of digits in the two integers, respectively.

Nonpolynomial-Time Algorithm

Definition: An algorithm is nonpolynomial-time if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

Example (Composite): The naive algorithm for determining whether n is composite compares n with the first $n - 1$ numbers to see if any of them divides n.

- Let $m = \log_2 n$ be the input size of this problem
- Thus, the complexity if $\Theta(n) = \Theta(2^{(\log_2 n)})$, which is $\Theta(2^m)$
- The algorithm is nonpolynomial!

Nonpolynomial-time algorithms are impractical.

- 2^n for $n = 100$: it takes billions of years!!!

In reality, an $O(n^{20})$ algorithm is not really practical.

The Class P

Definition: A problem is solvable in polynomial time (or more simply, the problem is in polynomial time) if there exists an algorithm which solves the problem in polynomial time

- This problem is called tractable.

Definition (The Class P): The class P consists of all decision problems that are solvable in polynomial time. That is, there exists an algorithm that will decide in polynomial time if any given input is a yes-input or a no-input.

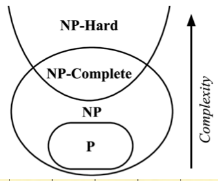
Question: How to prove that a decision problem is in P?

Answer: Find a polynomial-time algorithm.

Question: How to prove that a decision problem is not in P?

Answer: You need to prove that there is no polynomial-time algorithm for this problem. (much much harder)

- Some other definitions for potentially harder problems ....



Certificates and Verifying Certificates

Before introduce NP Problem, some new definitions ...

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

A certificate (or witness) is a specific object corresponding to a yes-input, such that it can be used to show that the input is indeed a yes-input.

Example (DKnapsack): Given V, is there a subset of the objects that fits in the knapsack and has total value at least V?

To show V is a yes-input, a certificate is a subset of the objects that

- fit in the knapsack (i.e., the sum weight does not exceed the capacity)
- have a total value at least V

A certificate (or witness) is a specific object corresponding to a yes-input, such that it can be used to show that the input is indeed a yes-input.

Verifying a certificate: Given a presumed yes-input and its corresponding certificate, by making use of the given certificate, we verify that the input is actually a yes-input.

The Class NP

Definition: The class NP consists of all decision problems such that, for each yes-input, there exists a certificate which allows one to verify in polynomial time that the input is indeed a yes-input.

NP – "nondeterministic polynomial-time"

Example (DKnapsack): Given V, is there a subset of the objects that fits in the knapsack and has total value at least V?

To show V is a yes-input, a certificate is a subset of the objects that

- fit in the knapsack (i.e., the sum weight does not exceed the capacity)
- have a total value at least V

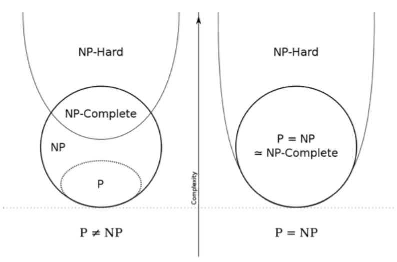
DKnapsack is an NP problem.

P=NP?

One of the most important problems in CS is

Whether P $=$ NP or P $\neq$ NP?

- Observe that $P \subseteq NP$.
- Intuitively, $NP \subseteq P$ is doubtful.



- NP-Hard: informally "at least as hard as the hardest problems in NP"
- NP-Complete: If the problem is in NP and all other NP problems are polynomial-time reducible to it.

However, we are still no closer to solving it.