Algorithm Efficiency

Luay Nakhleh
Computer Science
Rice University
Reading Material

- Chapter 3, Sections 2-3
Not All Correct Algorithms Are Created Equal

- We often choose the most efficient algorithm among the many correct ones.
- [In some cases, we might choose a slightly slower algorithm that’s easier to implement.]

Efficiency
- Time: How fast an algorithm runs
- Space: How much extra space the algorithm takes
- There’s often a trade-off between the two.
It’s All a Function of the Input Size

- We often investigate an algorithm’s complexity in terms of some parameter $n$ that indicates the input size.
- For an algorithm that sorts a list, $n$ is the number of elements in the list.
- For an algorithm that computes the cardinality of a set, $n$ is the number of elements in the set.
It’s All a Function of the Input Size

- In some cases, more than one parameter is needed to capture the input size.
- For an algorithm that computes the intersection of two sets A and B of sizes $m$ and $n$, respectively, the input size is captured by both $m$ and $n$. 
An algorithm that operates on graphs:

- If the graph is represented by its adjacency list, then the input size is often given by $m$ (number of edges) and $n$ (number of nodes).
- If the graph is represented by its adjacency matrix, then the input size is often given by $n$ alone (the number of nodes).
- However, it the choice of the input size depends on the algorithm and the assumptions on the input.
Consider the following simple algorithm for testing whether integer $p$ is prime:

- divide $p$ by every number between 2 and $p-1$; if the remainder is 0 in at least one case, return False, otherwise return True.

What is the input size?
Units for Measuring Running Time

- We'd like to use a measure that does not depend on extraneous factors such as the speed of a particular computer, the hardware being used, the quality of a program implementing the algorithm, or the difficulty of clocking the actual running time of the program.

- We usually focus on basic operations that the algorithm performs, and compute the number of times those basic operations are executed.
The Growth of Functions
• A difference in running times on small inputs is not what distinguishes efficient algorithms from inefficient ones.

• When analyzing the complexity of an algorithm, we pay special attention to the order of growth of the number of steps that the algorithm takes as the input size increases.
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

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

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

- \( n^2 = O(n^2) \)
- \( 3n^2 + 100 = O(n^2) \)
- \( n^2 \neq O(n) \)
- \( 5n^2 = O(n^3) \)
- \( 1000 = O(1) \)
- \( n \neq O(\log n) \)
EXAMPLE 7
In Section 4.1, we will show that $n^2 < 2^n$ whenever $n$ is a positive integer. Show that this inequality implies that $n$ is $O(2^n)$, and use this inequality to show that $\log n$ is $O(n)$.

Solution: Using the inequality $n^2 < 2^n$, we quickly can conclude that $n$ is $O(2^n)$ by taking $k = C = 1$ as witnesses. Note that because the logarithm function is increasing, taking logarithms (base 2) of both sides of this inequality shows that $\log n < n$. It follows that $\log n$ is $O(n)$. (Again we take $C = k = 1$ as witnesses.)

If we have logarithms to a base $b$, where $b$ is different from 2, we still have $\log_b n$ is $O(n)$ because $\log_b n = \log n / \log b < n / \log b$ whenever $n$ is a positive integer. We take $C = 1 / \log b$ and $k = 1$ as witnesses. (We have used Theorem 3 in Appendix 2 to see that $\log_b n = \log n / \log b$.)

As mentioned before, big-O notation is used to estimate the number of operations needed to solve a problem using a specified procedure or algorithm. The functions used in these estimates often include the following:

- $1$
- $\log n$
- $n$
- $n \log n$
- $n^2$
- $2^n$
- $n!$

Using calculus it can be shown that each function in the list is smaller than the succeeding function, in the sense that the ratio of a function and the succeeding function tends to zero as $n$ grows without bound. Figure 3 displays the graphs of these functions, using a scale for the values of the functions that doubles for each successive marking on the graph. That is, the vertical scale in this graph is logarithmic.
Let \( f(x) = a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0 \), where \( a_0, a_1, \ldots, a_{n-1}, a_n \) are real numbers. Then \( f(x) \) is \( O(x^n) \).

Suppose that \( f_1(x) \) is \( O(g_1(x)) \) and that \( f_2(x) \) is \( O(g_2(x)) \). Then \( (f_1 + f_2)(x) \) is \( O(\max(|g_1(x)|, |g_2(x)|)) \).

Suppose that \( f_1(x) \) is \( O(g_1(x)) \) and \( f_2(x) \) is \( O(g_2(x)) \). Then \( (f_1f_2)(x) \) is \( O(g_1(x)g_2(x)) \).
Of course, $17n^4 - 32n^2 + 5n + 107 = O(17n^4 - 32n^2 + 5n + 107)$.

If the running time of an algorithm is $f(x)$, the goal in using big-O notation is to choose a function $g(x)$ such that (1) $f(x) = O(g(x))$, and (2) $g(x)$ grows as slowly as possible so that to provide as tight an upper bound on $f(x)$ as possible.
Big-Omega

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

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

whenever $x > k$. [This is read as “$f(x)$ is big-Omega of $g(x)$.”]
Big-O and Big-Omega

\[ f(x) = \Omega(g(x)) \iff g(x) = O(f(x)) \]
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 $f$ 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.
Big-O, Big-Omega, and Big-Theta

\[ f(x) = \Theta(g(x)) \iff [f(x) = O(g(x)) \land f(x) = \Omega(g(x))] \]
Complexity of Algorithms
The complexity analysis framework ignores multiplicative constants and concentrates on the order of growth of the number of steps to within a constant multiplier for large-size inputs.
The complexity of a problem is not a statement about a specific algorithm for the problem. The complexity of an algorithm pertains to that specific algorithm.
Complexity: Problem vs. Algorithm

Example 1:

- When we say that the (comparison-based) sorting problem is solvable on the order of $n \log n$ algorithm, this means that there exists an algorithm for sorting that runs in $n \log n$ time.

- However, there are sorting algorithms that take more than $n \log n$ operations.
Complexity: Problem vs. Algorithm

Example 2:

When we say that a problem is NP-Complete, it means that we do not know of any polynomial-time algorithm for it.

However, there could be two algorithms for solving the problem, one that takes on the order of $2^n$ operations and the other takes on the order of $5^n$ operations.
Worst, Best, and Average Cases

- For an algorithm and input size $n$:
  - **Worst case**: The input of size $n$ that results in the largest number of operations.
  - **Best case**: The input of size $n$ that results in the smallest number of operations.
  - **Average case**: The expected number of operations that the algorithm takes on an input of size $n$. 
Worst, Best, and Average Cases

❖ Important: These cases are defined for a given input size \( n \), so you cannot change the size to get worst or best complexities.

❖ For example, in analyzing the running time of a graph algorithm and using input size \( n \) and \( m \), you cannot say the best case is when the graph has no edges, since that means you set \( m \) to 0 (this would be analogous to saying the best case of a sorting algorithm is an empty list!).
Worst, Best, and Average Cases

❖ The best-case of an algorithm is usually uninformative about the performance of the algorithm, but gives a lower bound (not necessarily tight) on the complexity of the algorithm.

❖ The average-case analysis is usually very hard to establish (we’ll see some examples when we cover discrete probability) and requires making assumptions about the input.

❖ The worst-case of an algorithm bounds the running time of the algorithm from above and gives an upper bound on the complexity of the problem. It is the most commonly used case in algorithm complexity analyses.
### Complexity of Algorithms

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(1)$</td>
<td>Constant complexity</td>
</tr>
<tr>
<td>$\Theta(\log n)$</td>
<td>Logarithmic complexity</td>
</tr>
<tr>
<td>$\Theta(n)$</td>
<td>Linear complexity</td>
</tr>
<tr>
<td>$\Theta(n \log n)$</td>
<td>Linearithmic complexity</td>
</tr>
<tr>
<td>$\Theta(n^b)$</td>
<td>Polynomial complexity</td>
</tr>
<tr>
<td>$\Theta(b^n)$, where $b &gt; 1$</td>
<td>Exponential complexity</td>
</tr>
<tr>
<td>$\Theta(n!)$</td>
<td>Factorial complexity</td>
</tr>
</tbody>
</table>

An algorithm has **polynomial complexity** if it has complexity $\Theta(n^b)$, where $b$ is an integer with $b \geq 1$. For example, the bubble sort algorithm is a polynomial-time algorithm because it uses $\Theta(n^2)$ comparisons in the worst case.

An algorithm has **exponential complexity** if it has time complexity $\Theta(b^n)$, where $b > 1$. The algorithm that determines whether a compound proposition in $n$ variables is satisfiable by checking all possible assignments of truth variables is an algorithm with exponential complexity, because it uses $\Theta(2^n)$ operations.

An algorithm has **factorial complexity** if it has $\Theta(n!)$ time complexity. The algorithm that finds all orders that a traveling salesperson could use to visit $n$ cities has factorial complexity; we will discuss this algorithm in Chapter 9.

### Tractability

A problem that is solvable using an algorithm with polynomial worst-case complexity is called **tractable**, because the expectation is that the algorithm will produce the solution to the problem for reasonably sized input in a relatively short time. However, if the polynomial in the big-$\Theta$ estimate has high degree (such as degree 100) or if the coefficients are extremely large, the algorithm may take an extremely long time to solve the problem. Consequently, that a problem can be solved using an algorithm with polynomial worst-case time complexity is no guarantee that the problem can be solved in a reasonable amount of time for even relatively small input values.

Fortunately, in practice, the degree and coefficients of polynomials in such estimates are often small.

The situation is much worse for problems that cannot be solved using an algorithm with worst-case polynomial time complexity. Such problems are called **intractable**. Usually, but not always, an extremely large amount of time is required to solve the problem for the worst cases of even small input values. In practice, however, there are situations where an algorithm with a certain worst-case time complexity may be able to solve a problem much more quickly for most cases than for its worst case. When we are willing to allow that some, perhaps small, number of cases may not be solved in a reasonable amount of time, the average-case time complexity is a better measure of how long an algorithm takes to solve a problem. Many problems important in industry are thought to be intractable but can be practically solved for essentially all sets of input that arise in daily life.

Another way that intractable problems are handled when they arise in practical applications is that instead of looking for exact solutions of a problem, approximate solutions are sought. It may be the case that fast algorithms exist for finding such approximate solutions, perhaps even with a guarantee that they do not differ by very much from an exact solution.

Some problems even exist for which it can be shown that no algorithm exists for solving them. Such problems are called **unsolvable** (as opposed to solvable problems that can be solved using an algorithm). The first proof that there are unsolvable problems was provided by the great English mathematician and computer scientist Alan Turing when he showed that the halting problem is unsolvable. Recall that we proved that the halting problem is unsolvable in Section 3.1. (A biography of Alan Turing and a description of some of his other work can be found in Chapter 13.)
Complexity of Algorithms

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>10</th>
<th>10^2</th>
<th>10^3</th>
<th>10^4</th>
<th>10^5</th>
<th>10^6</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>log n</td>
<td>3 × 10^{-11} s</td>
<td>7 × 10^{-11} s</td>
<td>1.0 × 10^{-10} s</td>
<td>1.3 × 10^{-10} s</td>
<td>1.7 × 10^{-10} s</td>
<td>2 × 10^{-10} s</td>
</tr>
<tr>
<td>n</td>
<td>10^{-10} s</td>
<td>10^{-9} s</td>
<td>10^{-8} s</td>
<td>10^{-7} s</td>
<td>10^{-6} s</td>
<td>10^{-5} s</td>
</tr>
<tr>
<td>n log n</td>
<td>3 × 10^{-10} s</td>
<td>7 × 10^{-9} s</td>
<td>1 × 10^{-7} s</td>
<td>1 × 10^{-6} s</td>
<td>2 × 10^{-5} s</td>
<td>2 × 10^{-4} s</td>
</tr>
<tr>
<td>n^2</td>
<td>10^{-9} s</td>
<td>10^{-7} s</td>
<td>10^{-5} s</td>
<td>10^{-3} s</td>
<td>0.1 s</td>
<td>0.17 min</td>
</tr>
<tr>
<td>2^n</td>
<td>10^{-8} s</td>
<td>4 × 10^{11} yr</td>
<td>10^{-5} s</td>
<td>10^{-3} s</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>n!</td>
<td>3 × 10^{-7} s</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

* more than 10^{100} years!
Algorithm 3: LinearSearch.

**Input:** An array $A[0 \ldots n-1]$ of integers, and an integer $x$.
**Output:** The index of the first element of $A$ that matches $x$, or $-1$ if there are no matching elements.

1. $i \leftarrow 0$
2. **while** $i < n$ and $A[i] \neq x$ **do**
   3. $i \leftarrow i + 1$
3. **if** $i \geq n$ **then**
   4. $i \leftarrow -1$
5. **return** $i$
Algorithm 4: MatrixMultiplication.

**Input:** Two matrices $A[0 \ldots n - 1, 0 \ldots k - 1]$ and $B[0 \ldots k - 1, 0 \ldots m - 1]$.

**Output:** Matrix $C = AB$.

1. for $i \leftarrow 0$ to $n - 1$ do
2.     for $j \leftarrow 0$ to $m - 1$ do
3.         $C[i, j] \leftarrow 0$;
4.         for $l \leftarrow 0$ to $k - 1$ do
6.     return $C$
Complexity of Algorithms

Algorithm 1: IsBipartite.

**Input:** Undirected graph $g = (V, E)$.

**Output:** True if $g$ is bipartite, and False otherwise.

1. **foreach** Non-empty subset $V_1 \subset V$ **do**
   2. $V_2 \leftarrow V \setminus V_1$;
   3. $\text{bipartite} \leftarrow \text{True}$;
   4. **foreach** Edge $\{u, v\} \in E$ **do**
      5. **if** $\{u, v\} \subseteq V_1$ or $\{u, v\} \subseteq V_2$ **then**
         6. $\text{bipartite} \leftarrow \text{False}$;
         7. **Break**;
      8. **if** $\text{bipartite} = \text{True}$ **then**
         9. **return** True;
   10. **return** False;
If there exists an polynomial-time worst-case algorithm for problem $P$, we say that $P$ is tractable.

If no polynomial-time worst-case algorithm exists for problem $P$, we say $P$ is intractable.
P vs. NP

- Tractable problems belong to class P.
- Problems for which a solution can be verified in polynomial time belong to class NP.
- Class NP-Complete consists of problems with the property that if one of them can be solved by a polynomial-time worst-case algorithm, then all problems in class NP can be solved by polynomial-time worst-case algorithms.
P vs. NP

❖ The million-dollar question: Is $P = NP$?
Questions?