CS2223: Algorithms
Introduction, Complexity, Notations and Recurrences

Algorithm: A tool to solve a well-defined *computational problem*. It takes some value, or set of values as **input** and produces some value, or set of values as **output**.

Examples: Algorithm to find the shortest way from your home to class, algorithm you use to quickly find a resource on the web, algorithm to determine how to schedule processes on your machine, algorithm to answer your database query, algorithm to sort a given set of numbers (objects), algorithm google uses to get data from different web sites etc, etc.

Many algorithms exist to solve a problem. How do you choose an "efficient one". Efficiency can be based on (a) time it takes to run an algorithm (**time complexity**) (b) amount of space required by the algorithm (**space complexity**).

An algorithm to a problem has three components (a) the actual algorithm written in pseudo-code (b) the proof of correctness of the algorithm (c) analysis of the algorithm (**time/space complexity**).

Analysis of algorithm – we can do best case analysis (the complexity for the best case input), worst case analysis (the complexity for the worst case input), average case analysis (the average complexity over all inputs). We typically focus on **worst case time complexity**.

For analysis of algorithms, we assume a Random Access Machine (RAM) model. The input is in binary; operations such as addition, multiply, divide etc on say 32-bit numbers can be done in unit time (constant time to be precise); instructions are executed one after the other.

1 Example: Computing the nth Fibonacci number (F_n)

Fibonacci Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
Consider the following algorithm to compute F_n:

fib1 (n):

Input: number n, for which F_n is to be computed

Output: F_n
if $n = 0$ then return 0;
if $n = 1$ then return 1;
return fib1 $(n-1)$ + fib1 $(n-2)$.

Proof of Correctness: This algorithm uses the definition, hence trivially correct.
Time Complexity Analysis: $T(n) = T(n-1) + T(n-2) + 3$
$T(n)$ denotes the time taken by the algorithm for input $= n$. It can be found to be exponential.
Below, we will show that the complexity is at least exponential.
$T(n) \geq 2T(n-2) + 3$
$T(n) \geq 2^2 T(n-4) + 2 * 3 + 3$
...
$T(n) \geq 3 * (1 + 2 + 2^2 + ... + 2^{n/2})$

Solving this, we get $T(n) = \Omega(1.4^n)$.

More precisely, we know that $T(0) = 1$; $T(1) = 2$. Using this, we get when n is odd, $T(n) \geq 5 * 2^{(n-1)/2} - 3$. When n is even, $T(n) \geq 4 * 2^{n/2} - 3$.

Similarly, use $T(n) \leq 2T(n-1) + 3$ to show that $T(n) = O(2^n)$.

More precisely, you can show that $T(n) \leq 5 * 2^{n-1} - 3$.

Another algorithm to compute F_n:
fib2 (n):
Input: number n, for which F_n is to be computed
Output: F_n
create array $f[0, 1, ..., n]$
$f[0] = 0$; $f[1] = 1$;
for $i = 2...n$ {
$f[i] = f[i-1] + f[i-2]$
}
return $f[n]$;

Proof of Correctness: After ith iteration, $f[0...i]$ are correctly computed, as

it has used the definition to compute it.
Time Complexity Analysis: it loops n times, and each loop, it does one addition. Therefore $T(n) = T(n-1) + 1$. This gives time complexity is $\Theta(n)$.

2 More about Analysis of Algorithms

We analyze the complexity when the input size is large, called **asymptotic analysis**. We also use notations like $O, \Omega, \theta, o, \omega$. We define them below.

Let $f(n)$ and $g(n)$ be two functions. We say that:

1. $f(n) = O(g(n))$ if f grows no faster than g. Mathematically, there exists a positive constant c, such that $f(n) \leq c.g(n)$ for all $n \geq n_0$.
2. $f(n) = \Omega(g(n))$ if f grows no slower than g. Mathematically, there exists a positive constant c, such that $f(n) \geq c.g(n)$ for all $n \geq n_0$.
3. $f(n) = \Theta(g(n))$ if f grows exactly as fast as g. Mathematically, there exists positive constants c_1, c_2, such that $c_1.g(n) \leq f(n) \leq c_2.g(n)$ for all $n \geq n_0$.
4. $f(n) = o(g(n))$ if f grows slower than g. Mathematically, for all positive constants c, $f(n) < c.g(n)$ for all $n \geq n_0$.
5. $f(n) = \omega(g(n))$ if f grows faster than g. Mathematically, for all positive constants c, $f(n) > c.g(n)$ for all $n \geq n_0$.

Note:
$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ if and only if $f(n) = \Theta(g(n))$
$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.
$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
$f(n) = O(g(n))$ and $g(n) = O(h(n))$ implies $f(n) = O(h(n))$, This is called the transitive property. Transitivity holds for Ω, Θ, o, and ω as well.
Examples:
$n^2 = \Theta(100n^2 + 1000)$
$n^2 = \omega(1000n + 10000)$

$log\ n = o(n)$
$log\ log\ n = o(log\ n)$
$log_2 n = \Theta(log_{10} n)$

$n^{1/2} = \omega(log\ n)$; Generally $n^\epsilon = \omega(log\ n)$, $\epsilon > 0$
$1.5^n = \omega(n^{1000})$; Generally $c^n = \omega(n^k)$, $c > 1$
$2^n = o(4^n)$, generally $c^n = o((c+\epsilon)^n)$

Let us try to show that $2^n = o(4^n)$.

We need to show that for every positive constant c, there exists n_0 such that $2^n < c.4^n$, for all $n \geq n_0$.

We know that if $c \geq 1$, the above is always true for all $n \geq 1$. Let us consider when $c < 1$, in which case let $c = 1/k$, where $k > 1$.

We need to find a n_0 such that $k * 2^n < 4^n$. We know that this is true for all $n > log\ k$.

Therefore for every $c < 1$ there exists a $n_0 = log 1/c$, such that $2^n < c.4^n$. Hence proved.

3 Some Basics

Exponentials:
$a^0 = 1$
$a^1 = a$
$a^{-1} = 1/a$
$(a^m)^n = (a^n)^m = a^{mn}$
$a^m a^n = a^{m+n}$

Logarithms:
$a = b^{log_b a}$
$log_c(ab) = log_c a + log_c b$
$log_b a^n = n log_b a$
$log_b a = \frac{log_c a}{log_c b}$
$log_b(1/a) = -log_b a$
$a^{log_b c} = c^{log_b a}$

Summations:

Arithmetic Series:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Geometric Series:

$$\sum_{k=0}^{n-1} a * r^k = a + a.r + a.r^2 + \ldots + a.r^{n-1} = \frac{a(r^n - 1)}{r - 1}, if r > 1$$

$$\sum_{k=0}^{n-1} a * r^k = a + a.r + a.r^2 + \ldots + a.r^{n-1} = \frac{a(1 - r^n)}{1 - r}, if r < 1$$

$$\sum_{k=0}^{\infty} a * r^k = \frac{a}{1 - r}, if r < 1$$

4 Recurrences

While analyzing complexity, we often run into expressions such as $T(n) = T(n-1) + T(n-2)$ (for fib1). This is called a **recurrence**. A recurrence is an equation (or an inequality) that describes a function in terms of its value for smaller inputs.

We often need to solve a recurrence as part of analyzing the complexity. We will look at the *recursion-tree method* for solving recurrences. We will illustrate this method with examples.

Example: Solve the recurrence $T(n) = 2T(n/2) + c$, where c is a constant.

$$\begin{aligned}
T(n) &= 2T(n/2) + c \\
&= 2(2T(n/4) + c) + c = 2^2T(n/4) + 2.c + c \\
&= 2^2(2T(n/8) + c) + 2.c + c = 2^3T(n/8) + 2^2.c + 2.c + c \\
&\vdots \\
&= 2^{log_2 n}T(1) + c + 2.c + 2^2.c + \ldots + 2^{n-1}.c
\end{aligned}$$

As T(1) is a constant, we get, $T(n) = \Theta(n)$.

Example: Solve the recurrence $T(n) = 2T(n/2) + c.n$, where c is a constant.

$$\begin{aligned} T(n) &= 2T(n/2) + c.n \\ &= 2(2T(n/4) + c.n/2) + c = 2^2T(n/4) + c.n + c.n \\ &= 2^2(2T(n/8) + c.n/(2^2)) + c + c = 2^3T(n/8) + c.n + c.n + c.n \\ &\vdots \\ &= 2^{log_2 n}T(1) + c.n + c.n + \ldots + c.n \ (c.n \text{ appears } log_2(n) \text{ times.}) \end{aligned}$$

As T(1) is a constant, we get, $T(n) = \Theta(nlog\ n)$.

Example: Solve the recurrence $T(n) = T(n/3) + T(2n/3) + c.n$, where c is a constant. Show that $T(n) = \Theta(nlog\ n)$.

Example: Solve the recurrence $T(n) = 3T(n/4) + c.n^2$, where c is a constant. Show that $T(n) = \Theta(n^2)$.

5 Pseudo-Code Conventions

We recommend the following conventions to be used while writing pseudo-code.

- Start each algorithm defining the input and the output parameters.
- Declare any data structure before using – define data structures such as arrays, stacks, queues, strings etc, but you need not define integers, floats etc.
- Assignment statements can be written as $a = k$, where a takes the value as specified on the right hand side.
- Looping constructs include – for, while, repeat. You can use { } to define a block of pseudo-code.
- Control structures include if-then-else.
- Use // for comments.