1. What is Complexity?

Complexity refers to the rate at which the required storage or consumed time grows as a function of the problem size.

The absolute growth depends on the machine used to execute the program, the compiler used to construct the program, and many other factors.

We would like to have a way of describing the inherent complexity of a program (or piece of a program), independent of machine/compiler considerations.

Cont..

This means that we must not try to describe the absolute time or storage needed.

We must instead concentrate on a “proportionality” approach, expressing the complexity in terms of its relationship to some known function. This type of analysis is known as **asymptotic analysis.**

Asymptotic analysis

Asymptotic analysis is based on the idea that as the problem size grows, the complexity can be described as a simple proportionality to some known function. This idea is incorporated in the

1. .“Theta”
2. “Big O”,

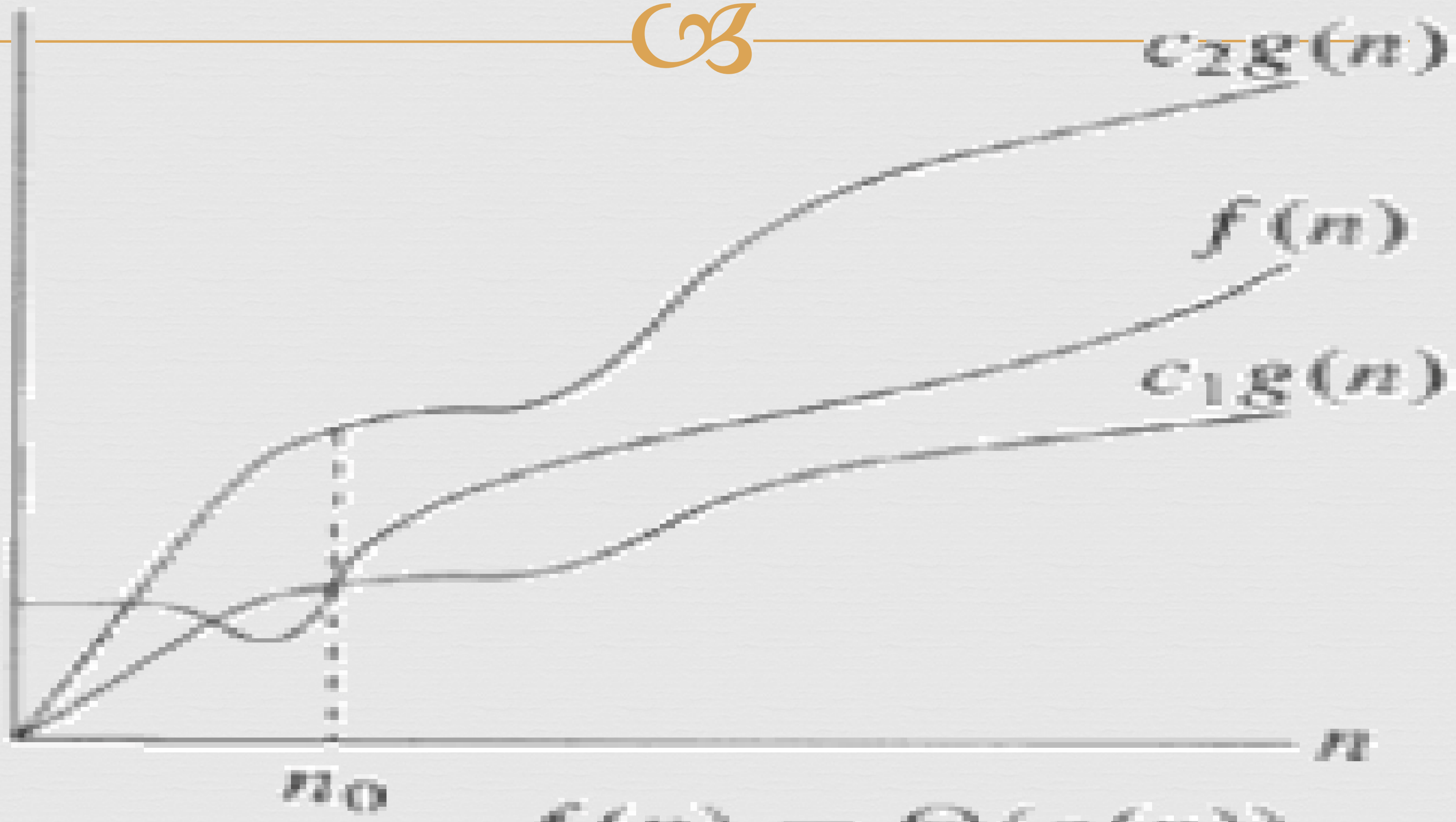
3 “Omega” and

3notation for asymptotic performance.

1.The Θ-Notation (Tight Bound) theta notation

This notation bounds a function to within constant factors. We say $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$, both inclusive. The *Figure below* gives an idea about function $f(n)$ and $g(n)$ where $f(n) = \Theta(g(n))$. We will say that the function $g(n)$ is asymptotically tight bound for $f(n)$.

Cont..



$$f(n) = \Theta(g(n))$$

2. The big O notation (Upper Bound) Big O" notation

This notation gives an upper bound for a function to within a constant factor. *Figure below* shows the plot of $f(n) = O(g(n))$ based on big O notation. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$.

Cont..

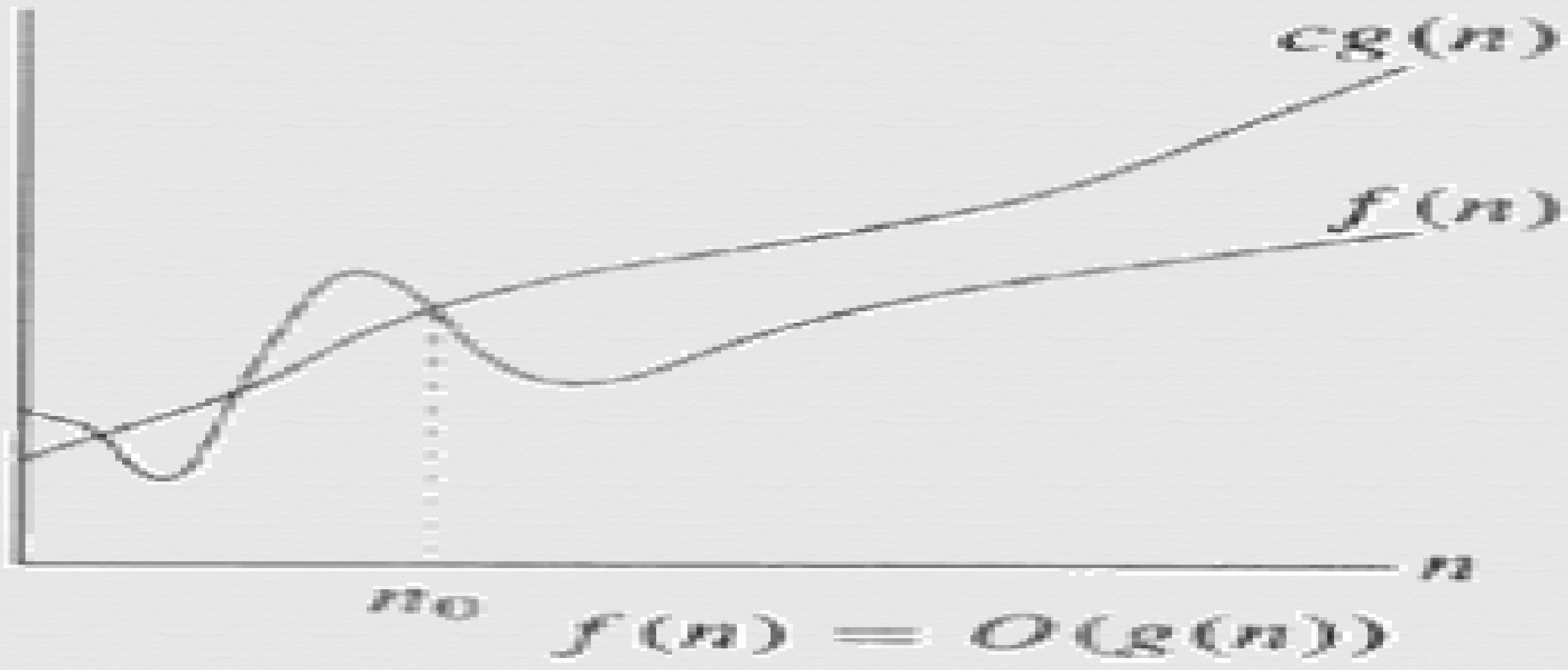


Figure 1.2: Plot of $f(n) = O(g(n))$

3. The Ω-Notation (Lower Bound) "Omega"

This notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$, if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$. *Figure 1.3* depicts the plot of $f(n) = \Omega(g(n))$.

Cont..

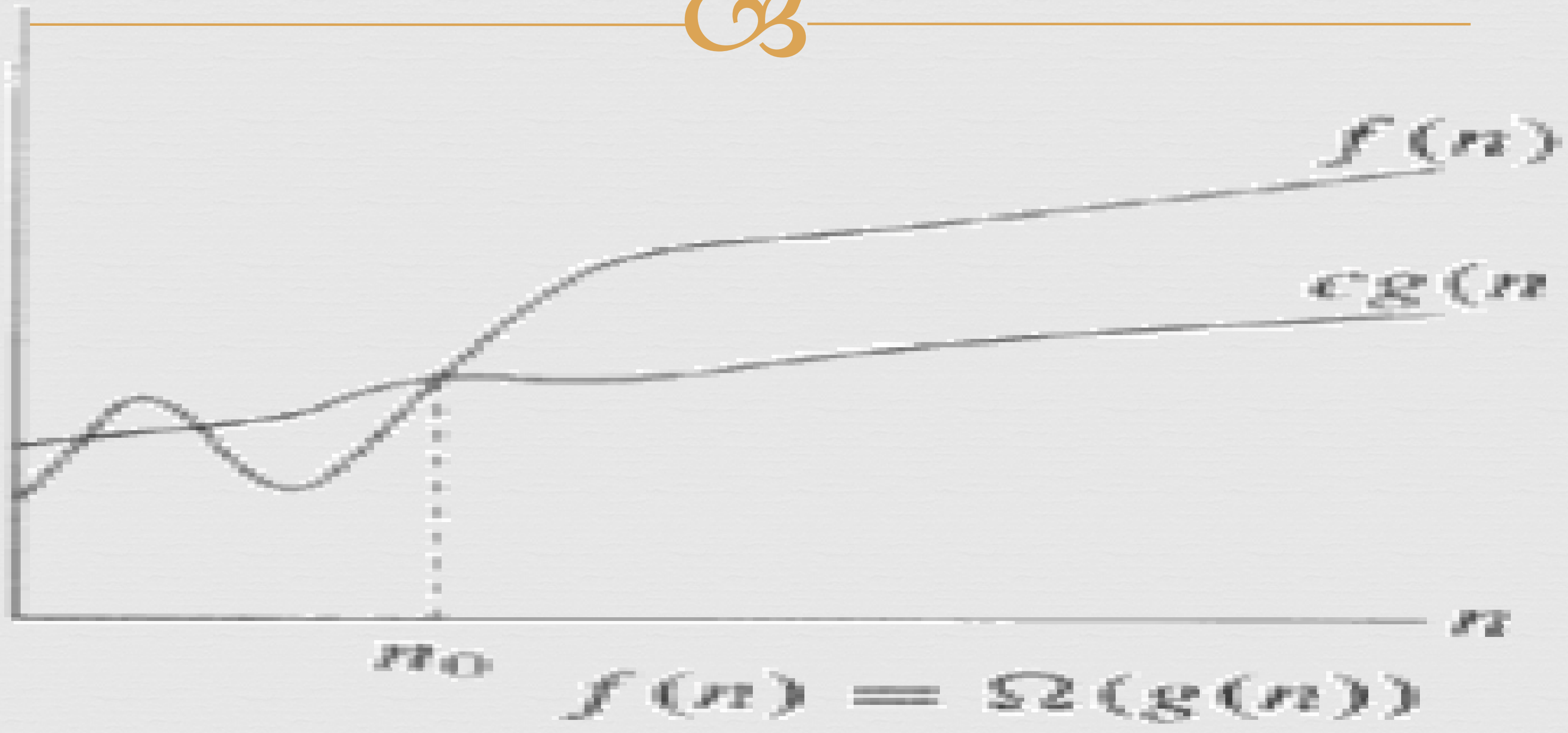


Figure 1.3: Plot of $f(n) = \Omega(g(n))$

model

- In order to analyze algorithm in a formal frameworks, we need a model of computational. The model is basically a normal computer, in which instructions executed sequentially. In models we have standard simple instructions such as additional, multiplication, comparison, and assignments, but unlike the case with real computers, it takes exactly one time unit to do anything.

Running time calculations

Example

```
        int sum (int n)
          {
            int i, partialsum;
1           partialsum=0;                 1unit
2            for(i=0;i<-n;i++)
3             partialsum+=i*i*i;          4units its goes up to n so 4n
4           return partialsum;            1unit
          }
```

Cont..

The analysis of above program is simple. The declaration count for no times. Line 1 and 4 count for one unit each. Line 3 counts for four units per time executed(two multiplication, one addition, and one assignments) and is executed N times for total 4N. Line 2 has the hidden costs of initializing i, testing i<=N, tests, and N for all the increments, which is 2N+2. we ignore the cost of calling function and returning, for total of

$4N+2+2N+2 = 6N +4$

Total time is 6N+4 units.

THANK YOU