CHAPTER 9: ASYMPTOTIC ANALYSIS

BIBHA STHAPIT
ASST. PROFESSOR
IoE, PULCHOWK CAMPUS

Complexity analysis

- Why we should analyze algorithms?
  - Predict the resources that the algorithm requires
    - Computational time (CPU consumption)
    - Memory space (RAM consumption)
    - Communication bandwidth consumption
  - The running time of an algorithm is:
    - The total number of primitive operations executed (machine independent steps)
    - Also known as algorithm complexity

Time complexity

- Worst-case
  - An upper bound on the running time for any input of given size
- Average-case
  - Assume all inputs of a given size are equally likely
- Best-case
  - The lower bound on the running time

- For example: Sequential search in a list of size n
  - Worst-case: n comparisons
  - Best-case: 1 comparison
  - Average-case: n/2 comparisons

Time-space trade-off

- A time space tradeoff is a situation where the memory use can be reduced at the cost of slower program execution (and, conversely, the computation time can be reduced at the cost of increased memory use).

- As the relative costs of CPU cycles, RAM space, and hard drive space change—hard drive space has for some time been getting cheaper at a much faster rate than other components of computers[citation needed]—the appropriate choices for time space tradeoff have changed radically.

- Often, by exploiting a time space tradeoff, a program can be made to run much faster.

Asymptotic notation

- Algorithm complexity is rough estimation of the number of steps performed by given computation depending on the size of the input data

  - Measured through **asymptotic notation**
    - O(g) where g is a function of the input data size

- Asymptotic means the line that tends to converge to a curve which may/may not eventually touch the curve but stays within bounds.

Asymptotic notation

- Asymptotic notation analyses the time complexity of an algorithm

- It is simply a function f(n) where n is the input size.

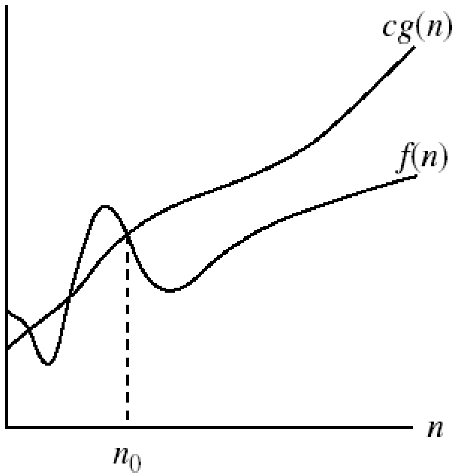
- Using the order of growth, we examine how fast the running time of algorithm increases when input size increases.

  - Examples:

    Linear complexity $O(n)$ – all elements are processed once (or constant number of times)

    Quadratic complexity $O(n^2)$ – each of the elements is processed n times

Big –O notation



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

- **Definition:** f(n) = O(g(n)) iff there are two positive constants c and n_0 such that

  $$|f(n)| \le c\,|g(n)| \text{ for all } n \ge n_0$$

- If f(n) is non-negative, we can simplify the last condition to

  $$0 \le f(n) \le cg(n) \text{ for all } n \ge n_0$$

- We say that “f(n) is big-O of g(n).”
- As n increases, f(n) grows no faster than g(n).
- In other words, g(n) is an *asymptotic upper bound* on f(n).

Big –O notation

- The running time is $O(n^2)$ means there is a function $f(n)$ that is $O(n^2)$ such that for any value of n, no matter what particular input of size n is chosen, the running time of that input is bounded from above by the value $f(n)$.
  - $3 * n^2 + n/2 + 12 \in O(n^2)$
  - $4 * n * \log_2(3 * n + 1) + 2 * n - 1 \in O(n * \log n)$

Show that $4n^2 = O(n^3)$.

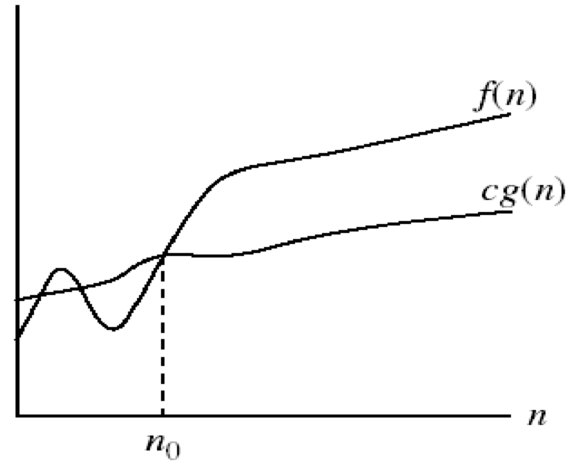
- **By definition,** we have $0 \le f(n) \le cg(n)$
- Substituting $4n^2$ as f(n) and n^3 as g(n),
- we get , $0 \le 4n^2 \le cn^3$
- Dividing by n^3 , $0/n^3 \le 4n^2/n^3 \le cn^3/n^3$
- $0 \le 4/n \le c$
- **Now to determine the value of c,**
- As $n \to \infty$, $4/n=0$
- we see that $4/n$ is maximum when $n=1$.
- Therefore, $c=4$.
- **To determine the value of n_0,**
- $0 \le 4/n_0 \le 4$
- $0 \le 4/4 \le n_0$
- $0 \le 1 \le n_0$ This means $n_0=1$.
- Therefore, $0 \le 4n^2 \le n^3 \; \forall \; n \ge n_0=1$.

Prove that : $n^2 + n = O(n^3)$

- **Proof:**
- Here, we have $f(n) = n^2 + n$, and $g(n) = n^3$
- Notice that if $n \geq 1$, $n \leq n^3$ is clear.
- Also, notice that if $n \geq 1$, $n^2 \leq n^3$ is clear.
- $n^2 + n \leq n^3 + n^3 = 2n^3$
- We have just shown that

  $n^2 + n \leq 2n^3$ for all $n \geq 1$.
- Thus, we have shown that $n^2 + n = O(n^3)$
- (by definition of Big-O, with $n_0 = 1$, and $c = 2$.)

Ω notation



g(n) is an ***asymptotic lower bound*** for f(n).

- Definition: $f(n) = \Omega(g(n))$ iff there are two positive constants c and n_0 such that

$$|f(n)| \geq c\ |g(n)| \text{ for all } n \geq n_0$$

- If $f(n)$ is nonnegative, we can simplify the last condition to

$$0 \leq c\ g(n) \leq f(n) \text{ for all } n \geq n_0$$

- We say that "*f(n)* is omega of *g(n)*."
- As n increases, *f(n)* grows no slower than *g(n)*.In other words, *g(n)* is an *asymptotic lower bound* on *f(n)*.

Ω notation

- When we say that the running time (no modifier) of an algorithm is $\Omega(g(n))$.

- we mean that no matter what particular input of size n is chosen for each value of n, the running time on that input is at least a constant times g(n), for sufficiently large n.

- $n^3 + 20n \in \Omega(n^2)$

Prove that : $n^3 + 4n^2 = \Omega(n^2)$

- **Proof:**
- Here, we have $f(n) = n^3 + 4n^2$, and $g(n) = n^2$.
- It is not too hard to see that if $n \geq 0$, $n^3 \leq n^3 + 4n^2$.
- We have already seen that if $n \geq 1$, $n^2 \leq n^3$.
- Thus when $n \geq 1$, $n^2 \leq n^3 \leq n^3 + 4n^2$.
- Therefore, $1n^2 \leq n^3 + 4n^2$ for all $n \geq 1$
- Thus, we have shown that $n^3 + 4n^2 = \Omega(n^2)$ (by definition of Big-Ω, with $n_0 = 1$, and $c = 1$).

Show: $3n^2 + n = \Omega(n^2)$

$0 \leq cg(n) \leq f(n)$

$0 \leq cn^2 \leq 3n^2 + n$

$0/n^2 \leq cn^2/n^2 \leq 3n^2/n^2 + n/n^2$

$0 \leq c \leq 3 + 1/n$ $\qquad$ $3+1/n = 3$

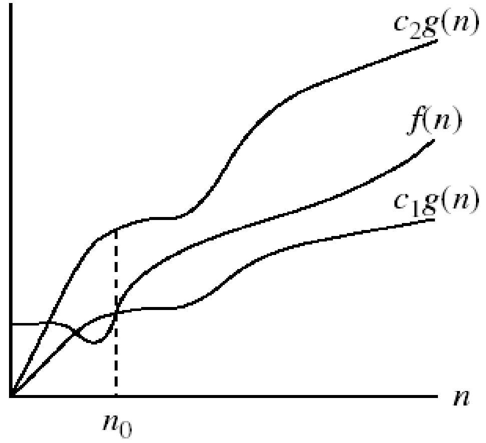
$0 \leq c \leq 3$ $\qquad$ $c = 3$

$0 \leq 3 \leq 3 + 1/n_0$

$-3 \leq 3-3 \leq 3-3 + 1/n_0$

$-3 \leq 0 \leq 1/n_0$ $\qquad$ $\lim_{n\to\infty}$ $n_0=1$ satisfies $1/n = 0$

Θ notation



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

- Definition: *f(n) = Θ(g(n))* iff there are three positive constants *c1, c2* and n_0 such that

  $c_1g(n) \le f(n) \le c_2g(n)$ for all $n \ge n_0$

- If *f(n)* is nonnegative, we can simplify the last condition to

  $0 \le c_1g(n) \le f(n) \le c_2g(n), \forall\, n \ge n_0$

- We say that "f(n) is theta of g(n)."

- As n increases, f(n) grows at the same rate as g(n). In other words, g(n) is an *asymptotically tight bound on f(n).*

Show : $n^2 + 5n + 7 = \Theta(n^2)$

- **Proof:**
- When $n \geq 0$, $n^2 \leq n^2 + 5n + 7$
- When $n \geq 1$, $n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2$
- Thus, when $n \geq 1$, $1n^2 \leq n^2 + 5n + 7 \leq 13n^2$
- Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$
- (by definition of Big- Θ, with $n_0 = 1$, $c1 = 1$, and $c2 = 13$.)

Show that $n^2/2 - 2n = \Theta(n^2)$

- By the definition, we can write,
  - $c1g(n) \leq f(n) \leq c2g(n)$
  - $c1n^2 \leq n^2/2 - 2n \leq c2n^2$
- Dividing by n^2, we get,
  - $c1n^2/n^2 \leq n^2/2n^2 - 2n/n^2 \leq c2\ n^2/n^2$
  - $c1 \leq 1/2 - 2/n \leq c2$
- This means $c2 = 1/2$ because $\lim n \to \infty\ 1/2 - 2/n = 1/2$ (Big O notation)
- To determine c1 using Ω notation, we can write ,
  - $0 < c1 \leq 1/2 - 2/n$
- We see that $0 < c1$ is minimum when $n = 5$. Therefore,
  - $0 < c1 \leq 1/2 - 2/5$
- Hence, $c1 = 1/10$
- Now let us determine the value of n_0
  - $1/10 \leq 1/2 - 2/n_0 \leq 1/2$
  - $2/n_0 \leq 1/2 - 1/10 \leq 1/2$
  - $2/n_0 \leq 2/5 \leq 1/2$
  - $n_0 \geq 5$
- You may verify this by substituting the values as shown below.
  - $c1n^2 \leq n^2/2 - 2n \leq c2n^2$
  - $c1 = 1/10$, $c2 = 1/2$ and $n_0 = 5$
  - $1/10(25) \leq 25/2 - 20/2 \leq 25/2$
  - $5/2 \leq 5/2 \leq 25/2$
- Thus, in general, we can write, $1/10n^2 \leq n^2/2 - 2n \leq 1/2n^2$ for $n \geq 5$.

Arithmetic of Big-O, Ω , and Θ notations

- Transitivity:
  - $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$) => $f(n) \in O(h(n))$
  - $f(n) \in (g(n))$ and $g(n) \in (h(n))$) => $f(n) \in (h(n))$
- Scaling: if $f(n) \in O(g(n))$ then for any $k > 0$, $f(n) \in O(kg(n))$
- Sums: if $f1(n) \in O(g1(n))$ and $f2(n) \in O(g2(n))$ then $(f1 + f2)(n) \in O(\max(g1(n), g2(n)))$

Basic rules

1. Nested loops are multiplied together.
2. Sequential loops are added.
3. Only the largest term is kept, all others are dropped.
4. Constants are dropped.
5. Conditional checks are constant (i.e. 1).

Example 1

```
//linear
for(i = 1 to n)
 {
         display   i;

}
```

- Ans: O(n)

Example 2

```
//quadratic
for(int i = 0; i < n;i++) {
        for(int j = 0; j < n; j++){
                //do swap stuff, constant time
        }
}
```

- A n s O(n^2)

Example 3

```
for(int i = 0; i < 2*n;i++) {
        cout << i << endl;
}
```

- A t first you might say that the upper bound is O(2n); however, we drop constants so it becomes O(n)

Example 4

```
//linear
for(int i = 0; i < n;i++)  {
        cout << i << endl;
}

//quadratic
for(int i = 0; i < n;i++)  {
        for(int j = 0; j < n; j++){
                //do constant time stuff
        }  }
```

- A n s : In this case we add each loop's Big O, in this case n+n^2. O(n^2+n) is not an acceptable answer since we must drop the lowest term. The upper bound is O(n^2). Why? Because it has the largest growth rate

Example 5

```
for(int i = 0; i < n;i++) {
        for(int j = 0; j < 2; j++){
                //do stuff
        }
}
```

- Ans: Outer loop is 'n', inner loop is 2, this we have 2n, dropped constant gives up O(n)

Example 6

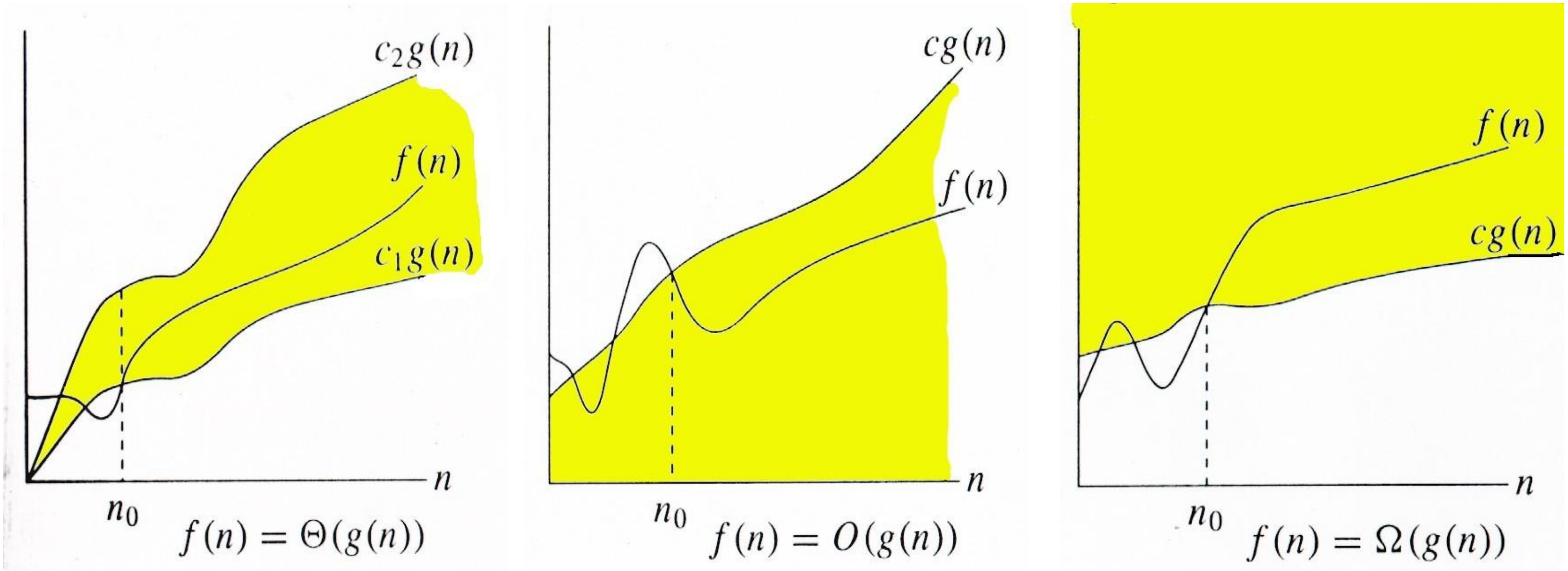
```
for(int i = 1; i < n; i *= 2) {
        cout << i << endl;
}
```

- There are n iterations, however, instead of simply incrementing, 'i' is increased by 2*itself each run. Thus the loop is O(log(n)).

Example 7

- for(int j = 1; j < n; j *= 2){ // log (n)
- for(int i = 0; i < n; i++) { //linear
- //do constant time stuff
- }
- }
- Ans: n*log(n)

Relation between Θ, O and Ω



THANK YOU!