[Mathematical Foundation]

Design and Analysis of Algorithms (CSc 523)

Samujjwal Bhandari
Central Department of Computer Science and Information Technology (CDCSIT)
Tribhuvan University, Kirtipur,
Kathmandu, Nepal.

Since mathematics can provide clear view of an algorithm. Understanding the concepts of mathematics aid in the design and analysis of good algorithms. Here we present some of the mathematical concepts that are helpful in our study.

Exponents

Some of the formulas that are helpful are :

$x^a x^b = x^{a+b}$

$x^{a\,/}x^b = x^{a-b}$

$(x^a)^b = x^{ab}$

$x^n + x^n = 2x^n$

$2^n + 2^n = 2^{n+1}$

Logarithms

Some of the formulas that are helpful are :

1. $\log_a{}^b = \log_c{}^b / \log_c{}^a$; $c>0$
2. $\log ab = \log a + \log b$
3. $\log a/b = \log a - \log b$
4. $\log (a^b) = b \log a$
5. Log $x < x$ for all $x>0$
6. Log $1 = 0$, $\log 2 = 1$, $\log 1024 = 10$.
7. $a^{\log_b n} = n^{\log_b a}$

Series

1. $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$
2. $\sum_{i=0}^{n} a^i \leq 1 / 1\text{-}a$; if $0<a<1$

   $= a^{n+1} - 1 / a\text{-}1$; else

3. $\sum_{i=1}^{n} i = n(n+1) / 2$

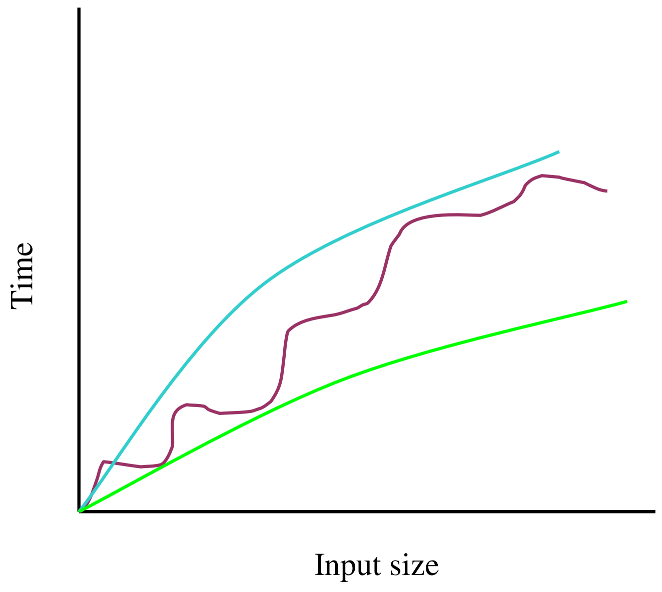
4. $\sum_{i=0}^{n} i^2 = n(n+1)(2n+1) / 6$

5. $\sum_{i=0}^{n} i^k \approx n^{k+1} / |k+1| ; k != -1$

6. $\sum_{i=1}^{n} 1/i \approx \log_e^n$

Asymptotic Notation

Complexity analysis of an algorithm is very hard if we try to analyze exact. we know that the complexity (worst, best, or average) of an algorithm is the mathematical function of the size of the input. So if we analyze the algorithm in terms of bound (upper and lower) then it would be easier. For this purpose we need the concept of asymptotic notations. The figure below gives upper and lower bound concept.



Why we concentrate on worst case mostly ? Why not best case or average case?

Big Oh (O) notation

When we have only asymptotic upper bound then we use O notation. A function f(x) = O(g(x)) (read as f(x) is big oh of g(x)) iff there exists two positive constants c and x_0 such that for all x >= x_0,
0 <= f(x) <= c*g(x)

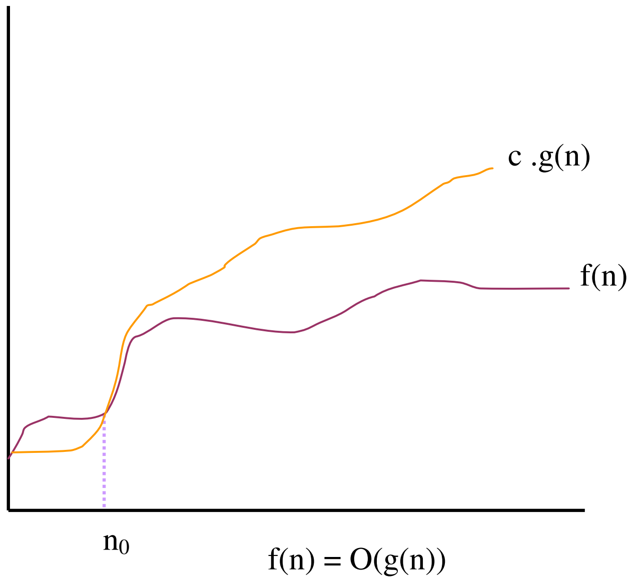
The above relation says that g(x) is an upper bound of f(x)

some properties:

Transitivity : f(x) = O(g(x)) & g(x) = O(h(x)) $\Rightarrow$ f(x) = O(h(x))

Reflexivity: f(x) = O(f(x))

O(1) is used to denote constants.



f(n) = O(g(n))

For all values of n >= n_0, plot shows clearly that f(n) lies below or on the curve of c*g(n)

Examples

1. $f(n) = 3n^2 + 4n + 7$
   $g(n) = n^2$, then prove that $f(n) = O(g(n))$.

   Proof: let us choose c and n_0 values as 14 and 1 respectively then we can have $f(n) <= c*g(n), n>=n0$ as
   $3n^2 + 4n + 7 <= 14*n^2$ for all $n >= 1$
   the above inequality is trivially true

   hence $f(n) = O(g(n))$

2. Prove that $n \log (n^3)$ is $O(\sqrt{n^3})$).

   Proof: we have $n \log (n^3) = 3n \log n$
   again, $\sqrt{n^3} = n \sqrt{n}$,
   if we can prove $\log n = O(\sqrt{n})$ then problem is solved
   because $n \log n = n O(\sqrt{n})$ that gives the question again.
   We can remember the fact that $\log^a n$ is $O(n^b)$ for all $a,b>0$.
   In our problem $a = 1$ and $b = ½$,
   hence $\log n = O(\sqrt{n})$.
   So by knowing $\log n = O(\sqrt{n})$ we proved that
   $n \log (n^3) = O(\sqrt{n^3})$).

3. Is $2^{n+1} =O(2^n)$?
   Is $2^{2n} = O(2^n)$?
   (See solution in the class)

Big Omega (Ω) notation

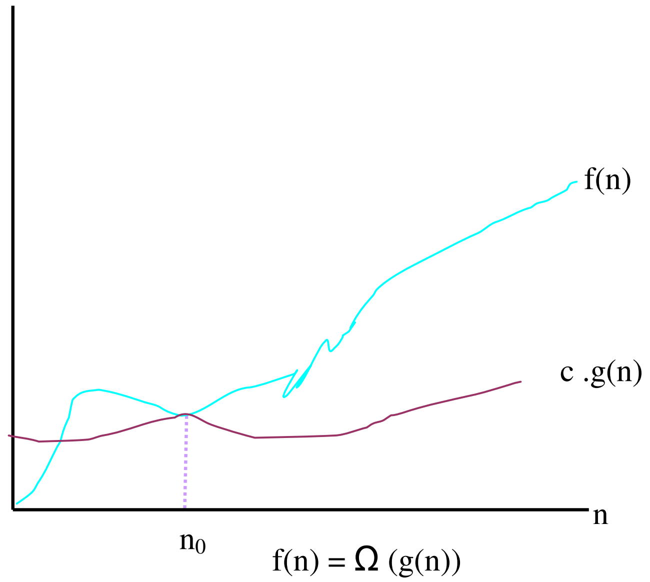
Big omega notation gives asymptotic lower bound. A function $f(x) = \Omega(g(x))$ (read as f(x) is big omega of g(x)) iff there exists two positive constants c and x_0 such that for all $x >= x_0$,
$0 <= c*g(x) <= f(x)$.

The above relation says that g(x) is an lower bound of f(x).

some properties:

Transitivity : $f(x) = O(g(x))$ & $g(x) = O(h(x)) \Rightarrow f(x) = O(h(x))$

Reflexivity: $f(x) = O(f(x))$



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

For all values of $n >= n_0$, plot shows clearly that f(n) lies above or on the curve of c*g(n).

Examples

1. $f(n) = 3n^2 + 4n + 7$

   $g(n) = n^2$, then prove that $f(n) = \Omega(g(n))$.

   Proof: let us choose c and n_0 values as 1 and 1, respectively then we can have

   $f(n) >= c*g(n)$, $n>=n0$ as

   $3n^2 + 4n + 7 >= 1*n^2$ for all $n >= 1$

   the above inequality is trivially true

   hence $f(n) = \Omega(g(n))$

Big Theta (Θ) notation

When we need asymptotically tight bound then we use notation. A function f(x) = (g(x)) (read as f(x) is big theta of g(x)) iff there exists three positive constants c_1, c_2 and x_0 such that for all $x >= x_0$,

$0 <= c_1*g(x) <= f(x) <= c_2*g(x)$

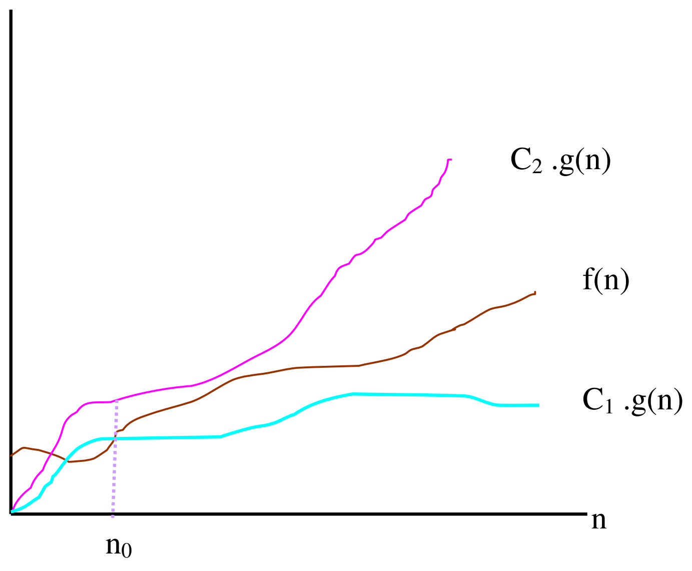
The above relation says that f(x) is order of g(x)

some properties:

Transitivity : $f(x) = \Theta(g(x))$ & $g(x) = \Theta(h(x)) \Rightarrow f(x) = \Theta(h(x))$

Reflexivity: $f(x) = \Theta(f(x))$

Symmetry: $f(x) = \Theta\ g(x)$ iff $g(x) = \Theta\ f(x)$



f(n) = Θ (g(n))

For all values of n >= n_0, plot shows clearly that f(n) lies between c_1* g(n)and c_2*g(n).

Examples

1. f(n) = $3n^2 + 4n + 7$

   g(n) = n^2 , then prove that f(n) = (g(n)).

   Proof: let us choose c1, c2 and n_0 values as 14, 1 and 1 respectively then we can have,

   f(n) <= c1*g(n), n>=n0 as $3n^2 + 4n + 7 <= 14*n^2$, and

   f(n) >= c2*g(n), n>=n0 as $3n^2 + 4n + 7 >= 1*n^2$

   for all n >= 1(in both cases).

   So c2*g(n) <= f(n) <= c1*g(n) is trivial.

   hence f(n) = Θ (g(n)).

2. Show $(n + a)^b = \Theta(n^b)$, for any real constants a and b, where b>0.

   Here, using Binomial theorem for expanding $(n + a)^b$, we get ,

$C(b,0)n^b + C(b,1)n^{b-1}a + \dots + C(b,b-1)na^{b-1} + C(b,b)a^b$

we can obtain some constants such that $(n + a)^b <= c_1*(n^b)$, for all $n >= n_0$

and

$(n + a)^b >= c_2*(n^b)$, for all $n >= n_0$

here we may take $c_1 = 2^b$ $c_2 = 1$ $n_0 = |a|$,

since $1 *(n^b) <= (n + a)^b <= 2^b *(n^b)$.

Hence the problem is solved.

Why $c_1 = 2^b$? since $\Sigma C(n,k) = 2^k$, where k=0 to n.

Little Oh (o) notation

Little oh (o) notation is used to denote the upper bound that is not asymptotically tight. A function f(x) = o(g(x)) (read as f(x) is little oh of g(x)) iff for any positive constant c there exists positive constant x_0 such that for all $x >= x_0$,

$0 <= f(x) < c*g(x)$

for example $4x^4$ is $O(x^4)$ but not $o(x^4)$.

Alternatively f(x) is little oh of g(x) if

$$\lim_{x->\infty} \frac{f(x)}{g(x)} = 0$$

Note: transitivity is satisfied.

Little Omega (ω) notation

Little omega (o) notation is used to denote the lower bound that is not asymptotically tight. A function f(x) = ω(g(x)) (read as f(x) is little omega of g(x)) iff for any positive constant c there exists positive constant x_0 such that for all $x >= x_0$,

$0 <= c*g(x) < f(x)$.

for example $x^3/7$ is $\omega(x^2)$ but not $\omega(x^3)$.

Alternatively f(x) is little omega of g(x) if

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = \infty$$

Note: transitivity is satisfied.

Complexity	10	20	30	40	50
n	0.00001 sec	0.00002 sec	0.00003 sec	0.00004 sec	0.00005 sec
n^2	0.0001 sec	0.0004 sec	0.0009 sec	0.016 sec	0.025 sec
n^3	0.001 sec	0.008 sec	0.027 sec	0.064 sec	0.125 sec
2^n	0.001 sec	1.0 sec	17.9 min	12.7 days	35.7 years
3^n	0.59 sec	58 min	6.5 years	3855 cent	2×10^8 cent

Some commonly used functions in algorithm analysis

f(x) = O(1)	constant
f(x) = C*log x	logarithmic
f(x) = C*x	linear
f(x) = C*x log x	linearithmic
$f(x) = C*x^2$	quadratic
$f(x) = C* x^3$	cubic
$f(x) = C* x^k$	polynomial in k
$f(x) = C* k^x$	exponential in k

Order of growth

$O(1) < C*\log x < C*x < C*x \log x < C*x^2 < C*x^3 < C*x^k < C*k^x$

Value of k should be in increasing order.

Before we start: Is the following statement true or false? $f(n) + g(n) = \Theta(\min(f(n),g(n)))$ Explain your answer.

Example Algorithm 1: Insertion Sort

Input: An array of numbers to be sorted A [] of length n.

Output: Sorted array A [].

Algorithm: InsertionSort (A [])

```
for (i=1; i<n; i++)
{
        x = A[i];
        //A[0]…A[i-1] is already sorted.
        j = i-1;
        while(j>=0 and A[j] > x)
        {
                A[j+1] = A[j];
                j = j-1;
        }
        A[j+1] = x;
}
```

Correctness:

The invariant of the outer loop that the array A[0] … A[i-1] is sorted contains no elements other than of original one. Finding invariant for inner loop is very hard still we can see that the inner loop is for insertion and always insert correctly.

The algorithm terminates when arrays last element is read by outer loop.

Efficiency

Time Complexity

	InsertionSort (A [])	cost	times
1.	for (i=1; i<n; i++)	c1	n
2.	{	~~c2~~	
3.	x = A[i]	c3	n-1
4.	//A[0]…A[i-1] is already sorted	~~c4~~	
5.	j = i-1;	c5	n-1
6.	while(j>=0 and A[j] > x)	c6	$\sum_{l=1}^{n} l$
7.	{	~~c7~~	
8.	A[j+1] = A[j];	c8	$\sum_{l=1}^{n} (l-1)$
9.	j = j-1;	c9	$\sum_{l=1}^{n} (l-1)$
10.	}	~~c10~~	
11.	A[j+1] = x;	c11	n-1
12.	}	~~c12~~	

Remember: l = i-1, cost and times are multiplied for each steps (not for space complexity). Strikethrough costs means the costs of no importance in our analysis.

Time complexity T(n) = c1*n + c3*(n-1) + c5*(n-1) + c6*$\sum_{i=1}^{n} l$ + c8*$\sum_{i=1}^{n} (l-1)$ + c9*$\sum_{i=1}^{n} (l-1)$ + c11*(n-1)

Best Case

Best case is achieved when the inner loop is not executed, for already sorted array this happens so best case analysis gives, T(n) = c1*n + c3*(n-1) + c5*(n-1) + c6*(n-1) + c11*(n-1)

$= \Theta(n)$.

Note: how some instances of inputs change the general behavior of an algorithm.

Worst Case

Worst case occurs when each execution of inner loop moves every element of an array.

$$T(n) = c1*n + c3*(n-1) + c5*(n-1) + c6*\frac{n(n+1)}{2} + c8*\frac{n(n-1)}{2} + c9*\frac{n(n-1)}{2} + c11*(n-1)$$

$$= \Theta(n^2)$$

Average Case

For average case inner loop inserts element from an array in the middle of an array, this takes $\Theta(i/2)$ time.

$$T_a(n) = \sum_{i=1}^{n-1} \Theta(i/2) = \Theta(\sum_{i=1}^{n-1} i) = \Theta(n^2)$$

Space Complexity

Space complexity is $\Theta(n)$ since space depends on the size of the input.

Example Algorithm 2: Merge Sort

Input: An array of numbers to be sorted A [] of length n.

Output: Sorted array A [].

Algorithm: MergeSort (A,l,h)

```
if(l<h)
{
    q = ⌊(l+h)/2⌋;
    MergeSort(A,l,q);
    MergeSort(A,q+1,h);
    Merge(A,l,q,h);
}
```

Algorithm: Merge (A,low,mid,high)

```
l = m =low; k = mid+1;
while((l<= mid) and k(<=high)
{
        if(A[l]<=A[k])
        {
                B[m] = A[l];
                l = l+1;
        }
        else
        {
                B[m] =A[k];
                k = k+1;
        }
        m = m+1;
}
if(l > mid)
        for(i =k; i < high; i++)
        {
                B[m] = A[i];
                m =m + 1;
        }
else
        for(i=l; i < mid; i++ )
        {
                B[m] = A[i];
                m = m + 1;
        }
for(j = low; j < high; j++)
        A[j] = B[j];
```

Correctness:

Correctness of a **MergeSort** can be verified by induction if we can prove that **Merge** is correct and correctness of **Merge** means two ranges of sorted sequence when get merged the result should be sorted and the content of the original input must be same. See, second loop just copies the array so if we prove that first loop produces sorted array then we are done. The invariants for the first loop are

Either m=1 or B[m] is sorted at every point in the algorithm, B[m-1] <= A[l] and B[m-1] <= A[k]. This property is true at the beginning and ending of loop (requires a bit effort to verify with all four conditions).

Efficiency

Time Complexity

To calculate the time complexity of the above MergeSort algorithm, first for the simplicity assume that the size of the input is power of 2. Now we can obtain time complexity as,

$T(n) = \Theta(1)$\{for if\} + $\Theta(1)$\{for 2nd statement\} + $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$ + T(Merge).

Here we need to calculate complexity of Merge algorithm also, complexity for this can be straightforwardly calculated as $\Theta(n)$ \{do yourself\}

Now, if n =1 then there is no process involved other than if (l<h) so $T(n) = \Theta(1)$

Floors and ceilings can be eliminated in asymptotic analysis. So for n>1 we can write

$T(n) = 2T(n/2) + \Theta(n)$

You can notice that we obtained valid recurrence relation with boundary condition and is expressed like

$T(n) = \Theta(1)$, when n = 1

$T(n) = 2T(n/2) + \Theta(n)$, when n > 1

We can write $n = 2^k$ (why? See our assumption) then,

$$
\begin{aligned}
T(n) &= 2T(2^{k-1}) + \Theta(2^k) \\
&= 2(2T(2^{k-2}) + \Theta(2^{k-1})) + \Theta(2^k) \\
&= 4T(2^{k-2}) + 2\Theta(2^{k-1}) + \Theta(2^k) \\
&\quad \ldots \\
&= 2^k\Theta(1) + 2^{k-1}\Theta(1) + \ldots + 2\Theta(2^{k-1}) + \Theta(2^k) \\
&= k\Theta(2^k) \\
&= \Theta(k\,2^k) \\
&= \Theta(n \log n)
\end{aligned}
$$

What if n is not a power of 2. Then there exists k such that $2^k < n < 2^{k-1}$ and we have

$$T(2^k) <= T(n) <= T(2^{k-1})$$

$$\Theta(k\,2^k) <= T(n) <= \Theta((k+1)\,2^{k+1})$$

here $\Theta((k+1)\,2^{k+1}) < 4k2^k$ so $\Theta((k+1)\,2^{k+1})$ is also $\Theta(k\,2^k)$. we have $k = \log n$ (not exactly but floors and ceilings can be omitted). So T(n) is $\Theta(n \log n)$.

Comparing performance

Comparing two sorting algorithms we can infer that MergeSort beats InsertionSort for larger enough data due to the fact that $\Theta(n \log n)$ grows more slowly than $\Theta(n^2)$. Since we are concerned with asymptotic analysis constants are neglected. However, to know at which point does merge sort beats insertion sort we need finer analysis. If we can do this then we can develop and design adaptive algorithms for e.g. By mixing both algorithms such that small data set uses insertion sort while the large data use merge sort.

After these materials all the detail analysis of most of the algorithms are avoided students can try for it (if interested).

Exercises

1. Show that 2^n is $O(n!)$.
2. Show that the time complexity of TOH is $O(2^n)$.
3.

   2.1.5: Prove theorem 2.1.

   Theorem 2.1: For any functions f(n) and g(n), $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

   2.1.6: Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst case running time is $O(g(n))$ and its best case running time is $\Omega(g(n))$.
4. $f(x) = a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$, where $a_0, a_2, \ldots, a_n$ are real numbers with $a_n != 0$. Then show that f(x) is $O(x^n)$, f(x) is $\Omega(x^n)$ and then show f(x) is order of x^n.