Discrete Mathematics 6/02/2019

Lecture 7: Order Notation

Instructor: Goutam Paul *Scribe: Nitin Prasad*

1 Family of Bachmann-Landau notations

There are family of notations invented by Paul Bachmann, Edmund Landau and others, collectively called Bachmann-Landau notation or asymptotic notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. Now let $f : \mathbb{N} \longrightarrow \mathbb{R}^+$ & $g : \mathbb{N} \longrightarrow \mathbb{R}^+$. Following are the members of family of Bachmann-Landau notations-

1. Small o/Small oh-We write $f \in o(g)$ if-

   $$\forall k > 0, \exists n_0 \; s.t. \forall n > n_0, f(n) < k \; g(n).$$

   The equivalent limit definition is given by

   $$\lim_{n \longrightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

   It describes that f is dominated by g asymptotically.

2. Big O/Big Oh-We write $f \in O(g)$ if-

   $$\exists k > 0, \exists n_0 \; s.t. \; \forall n > n_0, f(n) \leq k \; g(n)$$

   The equivalent limit definition is given by

   $$\limsup_{n \longrightarrow \infty} \frac{f(n)}{g(n)} < \infty.$$

   It describes that f is bounded above by g (upto constant factor) asymptotically.

3. Small Omega-We write $f \in \omega(g)$ if-

   $$\forall k > 0, \exists n_0 \; s.t. \forall n > n_0, f(n) > k \; g(n)$$

   The equivalent limit definition is given by

   $$\lim_{n \longrightarrow \infty} \frac{f(n)}{g(n)} = \infty.$$

   The third equivalent definition is given by-

   $$f \in \omega(g) \; if \; g \in o(f)$$

   It describes that f dominates over g asymptotically.

4. Big Omega-We write $f \in \Omega(g)$ if-

$$\exists k > 0 \& \exists n_0 \; s.t \; \forall n > n_0, f(n) \geq k \; g(n)$$

The equivalent limit definition is given by

$$\liminf_{n \longrightarrow \infty} \frac{f(n)}{g(n)} > 0$$

The third equivalent defination is given by

$$f \in \Omega(g) \; if \; g \in O(f)$$

It describes that f is bounded below by g asymptotically.

5. Big Theta- We write $f(n) \in \Theta(n)$ if-

$$\exists k_1 > 0, \exists k_2 > 0, \exists n_0 \; s.t. \forall n > n_0, k_1 \; g(n) \leq f(n) \leq k_2 \; g(n)$$

It can be defined using limits by-

$$0 < \liminf_{n \longrightarrow \infty} \frac{f(n)}{g(n)} \leq \limsup_{n \longrightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

We can alternatively define as-

$$f \in \Theta(g) \Longleftrightarrow f \in BigO(g) \; and \; f \in \Omega(g)$$

It describes that f is bounded both above and below by g asymptotically

6. Of the order of We write $f \sim g$ if-

$$\forall \epsilon > 0, \exists n_0 \; s.t. \; \forall n > n_0, \left| \frac{f(n)}{g(n)} - 1 \right| < \epsilon$$

The equivalent limit defination is given by-

$$\lim_{n \longrightarrow \infty} \frac{f(n)}{g(n)} = 1$$

It describes that f is equal to g asymptotically.

The table given in the following describes an analogy between the above notations and the commonly used inequality symbols.

Notation	Analogy
$f \in o(g)$	$<$
$f \in O(g)$	$\leq$
$f \in \omega(g)$	$>$
$f \in \Omega(g)$	$\geq$
$f \in \Theta(g)$	$\approx$
$f \sim g$	$=$

We often use the notations $f(n) = O(g(n))$ to infer $f \in O(g)$ even though there are many functions which lies in the set $O(g)$ and hence it clearly violates axiom of equality but still this notation is very common in use.

Following are the few properties of the above notation. Let f_1, f_2, g_1, g_2 be positive real valued functions defined over natural numbers.

1. If $f_1 \in o(g_1)$ and $f_2 \in o(g_2) \implies (af_1 + bf_2) \in o(ag_1 + bg_2)\ \forall a, b \geq 0$
   This property is also shared by the other notations.

2. If $f_1 \in o(g_1)$ and $f_2 \in o(g_2) \implies (f_1 \cdot f_2) \in o(g_1 \cdot g_2)$
   This property is also shared by the other notations.

3. If $f_1 \in o(g_1)$ and $f_2 \in O(g_2) \implies (f_1 \cdot f_2) \in o(g_1 \cdot g_2)$

2 Time complexity and analysis of algorithms

Time complexity of an algorithm describes the amount of time taken to run that algorithm.It is commonly estimated by counting the number of elementary operation performed by the algorithm assuming that each elementary operation takes a fixed amount to perform and hence the amount of time taken and the no of basic operations are taken to differ by at most a constant factor.

Since running time of an algorithm A depends depends on the different inputs I even if they are of the same size n therefore we consider the following three cases-

1. Worst case time complexity-The worst time complexity $W_A(n)$ is defined by-

$$W_A(n) = \max_I \Big\{ t(I) : I \ is \ input \ of \ size \ n \Big\}$$

   where $t(I)$ is the no of basic operations.

2. Best case time complexity-The best time complexity $W_A(n)$ is defined by-

$$W_A(n) = \min_I \Big\{ t(I) : I \ is \ input \ of \ size \ n \Big\}$$

3. Average case time complexity-It is defined by-

$$Avg_A(n) = \sum_{I \ is \ input \ of \ size \ n} t(I)\mathbb{P}(I)$$

   where $\mathbb{P}(I)$ is the probability of occurrence of input I.

Since computation of all the above function is quite difficult therefore one focuses on the asymptotic behaviour of these functions and express them using big O notation.

Definition 2.1. *Let $S \subseteq \mathbb{N}$, we define $f(n) \in O(g(n))$ on S if-*

$$\exists c > 0 \ \& \ \exists n_0 \in \mathbb{N} s.t.\ f(n) \leq c\, g(n) \ \ \forall n \geq n_0 \ \& \ n_0 \in S$$

Theorem 2.2. *Let $S = \{b^k, k \in \mathbb{N}\}$ where b is a fixed natural number. If f and g are monotonically increasing functions & $f(n) \in O(g(n))$ on S. If $g(b^r) \leq \lambda g(b^{r+1})\ \forall r \in \mathbb{N}$, then $f \in O(g)$*

Proof. Let $\forall k > k_0, f(b^k) \leq c\ g(b^k)$. Now for any n we consider k *s.t.* $b^k \leq m \leq b^{k+1}$

$$\begin{aligned} \therefore f(m) &\leq f(b^{k+1}) \\ &\leq cg(b^k) \\ &\leq c\lambda g(b^k) \\ &= c'g(m) \end{aligned}$$

□

2.1 Master Theorem

Theorem 2.3. *(Master theorem)This theorem is used for asymptotic analysis of recurrence relations which arise from time complexity of various divide and conquer algorithms.Now consider an algorithm which divides a problem of size n into $'a'$ no of sub-problems each of size $\left[\frac{n}{b}\right]$ and let $f(n)$ denotes the time to divide the problem and recombine the sub-problems.Here we assume a and b are real constants such that $a \geq 1$ & $b > 1$. Therefore $T(n)$ can be expressed using recurrence relations that takes the form-*

$$T(n) = aT\left(\left[\frac{n}{b}\right]\right) + f(n)$$

where $[x]$ is the greatest integer function.

Let $c_0 = \log_b a$ and now if-

1. *$f(n) \in O(n^c)$ where $c < c_0$ then $T(n) = \Theta(n^{c_0})$*
   This case describes that if time taken to split and recombine the problem is dominated by the arising sub-problems then the former one doesn't have significant contribution in total time.

2. *$f(n) = \Theta(n^{c_0} log^k n)$ for any $k \geq 0$ then $T(n) \in \Theta(n^{c_0} log^{k+1} n)$ This case deals with algorithms in which splitting time is comparable to sub-problems.*

3. *If $f(n) \in \Omega(n^c)$ where $c > c_0$ and $af\left(\left[\frac{n}{b}\right]\right) \leq kf(n)$ for some $0 < k < 1$ and $\forall n \geq n_0$ then $T(n) \in \theta(f(n))$*

Lemma 2.4. *We will first prove this theorem for values of n which are integral powers of b.*

Case1-*Let m_0 be such that $f(b^m) \leq \lambda b^{cm_0}\ \forall m > m_0$.Let $n = b^m$ and now we have-*

$$\begin{aligned}
T(n) &= aT\Big(\frac{n}{b}\Big) + f(n) \\
&= a^2T\Big(\frac{n}{b^2}\Big) + f(n) + f\Big(\frac{n}{b}\Big) \\
&= a^{\log_b n}T(1) + \sum_{i=0}^{m} a^i f\Big(\frac{n}{b^i}\Big) \\
&= n^{\log_b a}T(1) + \sum_{i=0}^{m_0} a^i f\Big(\frac{n}{b^i}\Big) + \sum_{i=m_0}^{m} a^i f\Big(\frac{n}{b^i}\Big) \\
&\leq n^{c_0}T(1) + \sum_{i=0}^{m_0} a^i \lambda\Big(\frac{n}{b^i}\Big)^{c_0-\epsilon} + C_1 \sum_{i=m_0}^{m} a^i \\
&= n^{c_0}T(1) + \lambda n^{c_0-\epsilon} \sum_{i=0}^{m-m_0} b^{i\epsilon} + C_1 a^m \frac{1-\frac{1}{a^{m_0}}}{1-\frac{1}{a}} \\
&= n^{c_0}T(1) + C_2 n^{c_0} + \lambda n^{c_0-\epsilon} \frac{b^{\epsilon(m-m_0+1)}-1}{b-1} \\
&\leq n^{c_0}T(1) + C_2 n^{c_0} + \lambda n^{c_0-\epsilon} \frac{b^{\epsilon(m-m_0+1)}}{b-1} \\
&= n^{c_0}T(1) + C_2 n^{c_0} + C_3 n^{c_0}
\end{aligned}$$

$\therefore T(n) \in O(n^{c_0})$

Case2-*Let m_0 be such that $f(b^m) \leq \lambda b^{mc_0}\log^k(b^m)\ \forall m \geq m_0$.Let $g(n) = n^{c_0}\log^k n$Now we have*

$$\begin{aligned}
T(n) &= aT\Big(\frac{n}{b}\Big) + f(n) \\
&= n^{c_0}T(1) + \sum_{i=0}^{m-m_0} a^i f\Big(\frac{n}{b^i}\Big) + \sum_{i=m-m_0+1}^{m} a^i f\Big(\frac{n}{b^i}\Big) \\
&\leq n^{c_0}T(1) + C_1 n^{c_0} + \lambda \sum_{i=0}^{m-m_0} a^i \Big(\frac{n}{b}\Big)^{c_0} \log^k\Big(\frac{n}{b^i}\Big) \\
&\leq n^{c_0}T(1) + C_1 n^{c_0} + \lambda \sum_{i=0}^{m} a^i \Big(\frac{n}{b}\Big)^{c_0} \log^k\Big(\frac{n}{b^i}\Big) \\
&= n^{c_0}T(1) + C_1 n^{c_0} + \lambda n^{c_0}\log^k n \sum_{i=0}^{m} \Big(1 - \frac{\log b^i}{\log n}\Big)^k \\
&= n^{c_0}T(1) + C_1 n^{c_0} + \lambda m n^{c_0}\log^k n \\
&= n^{c_0}T(1) + C_1 n^{c_0} + C_2 n^{c_0}\log^{k+1} n
\end{aligned}$$

$\therefore f(n) \in O(nlog^{k+1}n)$

Case3-*Let m_0 be such that $f(b^m) \geq \lambda b^{mc}\ \forall m \geq m_0$.Let $c = c_0 + \epsilon$*

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + f(n) \\ &= n^{c_0}T(1) + \sum_{i=0}^{m-m_0} a^i f\left(\frac{n}{b^i}\right) + \sum_{i=m-m_0+1}^{m} a^i f\left(\frac{n}{b^i}\right) \\ &\geq n^{c_0}T(1) + C_1 n^{c_0} + \lambda \sum_{i=0}^{m-m_0} a^i \left(\frac{n}{b}\right)^{c_0+\epsilon} \\ &\geq n^{c_0}T(1) + C_1 n^{c_0} + \lambda n^{c_0+\epsilon} \sum_{i=0}^{m} b^{-i\epsilon} \\ &\geq n^{c_0}T(1) + C_1 n^{c_0} + \lambda n^{c_0+\epsilon} \end{aligned}$$

$\therefore T(n) \in \Omega(n^c)$

Now since $af\left(\frac{n}{b}\right) \leq kf(n), \therefore$ we have-

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + f(n) \\ &= n^{c_0}T(1) + \sum_{i=0}^{m} a^i f\left(\frac{n}{b^i}\right) \\ &\leq n^{c_0}T(1) + \sum_{i=0}^{m} k^i n^c \\ &\leq n^{c_0}T(1) + \frac{1}{1-k} n^c \end{aligned}$$

$\therefore T(n) \in O(n^c)$
$\because T(n) \in \Omega(n^c)$ & $T(n) \in O(n^c)\ \ \therefore f(n) \in \Theta(n^c)$

2.1.1 Applications of master theorem

Following are the few recurrence relations as examples whose asymptotic behaviour can be described using master theorem

1. $T(n) = \sqrt{2}T\left(\frac{n}{2}\right) + \log n$
   Here $c_0 = \log_2 \sqrt{2} = \frac{1}{2}$
   $\because f(n) = \log(n) = O(n^{\frac{1}{2}})\ \ \therefore$ by case1 of mater theorem we have $T(n) = O(n^{\frac{1}{2}})$

2. $T(n) = \sqrt{2}T\left(\frac{n}{2}\right) + n$
   Now $c_0 = 1$ & $f(n) = n = O(n \log^0 n)\ \therefore$ by case2 of master theorem we have $T(n) = O(n \log n)$

3. $T(n) = 16T\left(\frac{n}{4}\right) + n!$
   Here $c_0 = 2$ & $f(n) = n!$. Now by Stirling's approximation we have $f(n) \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

and hence $f(n) = O(n!)$ and also $16\left(\frac{4n}{4}\right)! \leq \frac{2}{3}(4n)!\ \forall n \geq 1$
Hence by third case of master theorem we have $T(n) = O(n!)$

2.1.2 Inadmissible equations

Master theorem cannot be applied in the following cases-

1. $T(n) = 2^n T\left(\frac{n}{2}\right)$
   Here a is not a constant,i.e the no of subproblems are not fixed.

2. $T(n) = 0.5T\left(\frac{n}{2}\right) + n$
   Here $a < 1$ i.e the no of subproblem is less than 1 hence the master theorem doesn't apply here.

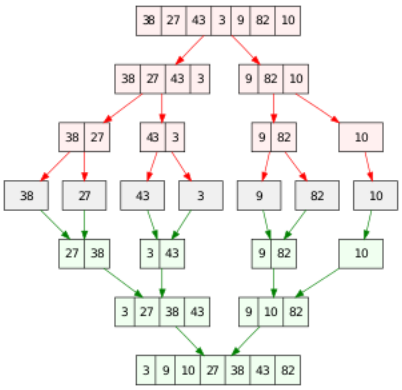
3. $T(n) = 64T\left(\frac{n}{2}\right) - n^2 \log n$
   Here f(n) is not positive hence master theorem cannot be applied.

4. $T(n) = T\left(\frac{n}{2}\right) + n(2 - \cos n)$
   Here $c_0 = 0$ and $f(n) = n(2 - cosn) \in \Omega(n)$.Now for case 3 to hold we need k¡0 such that eventually $f(n) \leq kf(2n)$ i.e. $\frac{(2-\cos n)}{2-\cos 2n} \leq 2k < 2$ but because $S = \{\cos n : n \in \mathbb{N}\}$ is dense in the set (-1,1) and hence we can assume a sequence $\{\cos a_i\}$ converging to -1 and hence $\left\{\frac{f(a_i)}{f(2a_i)}\right\}$ converges to 3 and hence there is no $k < 1$ which satisfy our required condition and hence master theorem cannot be applied here.

2.2 Calculating time complexity of algorithms

2.2.1 Mergesort

Mergesort is a divide and conquer algorithm which is used to sort a list of numbers. This is a recursive algorithm which takes an array as input then divides the array into two halves and calls itself again for sorting those two halves. This process continues till their size is 1 and we cannot divide them anymore. We assume that all array of size 1 is already sorted. We eventually merge them such that in every step after merging two already sorted array we again get a sorted array and eventually we get the final sorted array of initial size.



Pseudocode for Mergesort

We have MERGESORT(A,p,q) function which takes the given array and two positions on the array as its input.

```
MERGESORT(A,p,q)
1 if(p¡q)
2    pindex=⌈(p + q)/2⌉
3    MERGESORT(A,p,pindex)
4    MERGESORT(A,pindex+1,q)
5    MERGE(A,p,pindex,r)
```

Now following is the MERGE function that takes the initial array and three positions on the array as its input. MERGE(A,p,q,r)

```
1    m=q-p+1
2    n=r-q
3    let L[1 ··· m] and R[1 ··· n] be the two arrays
4    for i=0 to m-1
5        L[i]=A[p+i]
6    for j=0 to n-1
7        R[j]=A[q+j+1]
8    i=j=0
9    for k =p to r
10       if(i¡m and j¡n)
11           if L[i]≤R[j]
12               A[k]=L[i]
13               i=i+1
14           else A[k]=R[j]
15               j=j+1
16   for i¡m
17       array[i+j]=L[i]
18       i=i+1
19   for j¡n
20       array[i+j]=R[j]
21       j=j+1
```

Now from the above pseudocode it is clear that in worst case time taken to merge two sorted array of size n_1 and n_2 takes $n_1 + n_2 - 1 = n - 1$ no of steps. $\therefore$ If $T(n)$ denotes the time taken to sort an array of given length then then for all n of the form 2^k for some $k \in \mathbb{N}$ we have the following recursive algorithm.

$$T(n) = 2T\left(\frac{n}{2}\right) + f(n)$$

where $f(n) \in O(n)$. Now since $\log_2 2 = 1$, $\therefore$ by master theorem, $T(n) \in O(n \log n)$ on $S = \{2^k : k \in \mathbb{N}\}$

Now when n is not of the form 2^k then

$$T(n) = T\left\lfloor\frac{n}{2}\right\rfloor + T\left\lceil\frac{n}{2}\right\rceil + f(n)$$

Now $T(2) = 2T(1) + f(2) > T(1)$
Let for all $m < n_0$, $T(m) < T(m+1)$, $\therefore$ we have,

$$\begin{aligned} T(n_0+1) &= T\left\lfloor \frac{n_0+1}{2} \right\rfloor + T\left\lceil \frac{n_0+1}{2} \right\rceil + f(n_0+1) \\ &> T\left\lfloor \frac{n_0}{2} \right\rfloor + T\left\lceil \frac{n_0}{2} \right\rceil + f(n_0) \\ &= T(n) \end{aligned}$$

$\therefore$ By principle of strong induction we have $T(n+1) > T(n)$ and hence $T(n)$ is an increasing function and hence by Theorem2.2 we have $T(n) \in O(n \log n)$

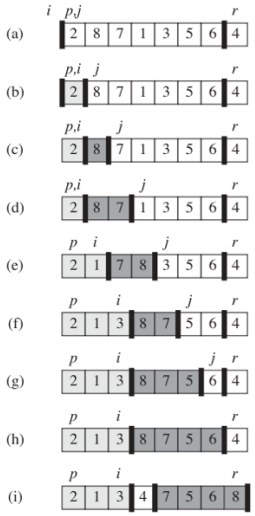
2.2.2 Quicksort

Quicksort is also a divide and conquer algorithm which is again a recursive algorithm that sorts an array of numbers.Following are three process involved in sorting a typical array $A[p \cdots r]$

- **Divide:** The initial array $A[p \cdots r]$ which itself may be part of an array is divided into two subarrays given by $A[p \cdots q-1]$ and $A[q+1 \cdots r]$ partitioned by the A[q] element such that all the elements in the array $A[p \cdots q-1]$ is less than A[q] which itself is less than all the elements present in array $A[q+1 \cdots]$.This is implemented by choosing any arbitrary element of the subarray for which we generally choose the last element of the array and is named as pivot. We then start to check all the elements from the left end which are less than pivot and then start to accumulate them at the left portion of the subarray.This process continues till we reach the element present in A[r-1] and then we change can finally swap the A[r] element with some other element of the subarray such that all the elements in the left of new position of pivot is lesser than pivot and the ones greater than pivot are larger than pivot.
- **Conquer:**The two subarrays are then sorted by the recursive call to quicksort.
- **Combine:**There is no work needed in combining as once all the subarrays are sorted, the entire array $A[p \cdots r]$ is now sorted.

Pseudocode of quicksort function

```
Quicksort(A,p,r)
1   if(p¡r)
2   q=Partition(A,p,r)
3   Quicksort(A,p,q-1)
4   Quicksort(A,q+1,r)
```



Now the Partition(A,p,r) is given by-

```
Partition(A,p,r)
1   x=A(r)
2   i=p-1
3   for j=p to r-1
4       if A[j]≤ x
5                   i=i+1
6                   exchange A[i] with A[j]
7   exchange A[i+1] with A[r]
8   return i+1
```

Time complexity of Quicksort

1. **Worst case-** In worst case, time taken by partition function which is $f(n)$ takes linear time $f \in \Theta(n)$. Moreover in worst case, the index returned by the partition function is one of the boundary index and hence we have $T(n) = T(n-1) + T(0) + f(n)$ and by this recurrence relation it can be shown that $T(n) \in \theta(n^2)$.

2. **Best case-** In best case at eash step, size of initial $subarray(n)$ is reduced by half i.e. $\frac{n}{2}$ or one of $\lfloor \frac{n}{2} \rfloor$ and other of $\lceil \frac{n}{2} \rceil$

   Hence when n is of the form 2^k we have the recurrence relation given by-

$$T(n) = 2T\left(\frac{n}{2}\right) + f(n)$$

   Hence by similar arguments as in the case of mergesort we have $T(n) \in O(n \log n)$

2.3 Lower bound of comparison based sorting

Theorem 2.5. *If A be a deterministic comparison based sorting algorithm and if $T(n)$ denote its time complexity in worst case then $T(n) \in \Omega(n \log n)$*

Proof. Given an n input of n elements there are $n!$ different permutations of given input and with only one correct permutation which is to be given as output. Let S be the set of all permutations. Now every comparison step reduces the number of possible candidates for output,$\therefore$ let S_n denotes the set of permutations which are consistent after n comparisons. Now $(k+1)^{th}$ comparison step can at most reduce the no of possible candidates for output by half of the initial value i.e.

$$\begin{aligned} |S(k+1)| &\geq \frac{|S(k)|}{2} \quad for k \geq 1 \\ \implies S(k) &\geq \frac{n!}{2^k} \end{aligned}$$

Now to get the desired permutation we must perform k_0 steps such that $|S(k_0)| = 1$ and hence $k_0 \geq \log_2 n!$. Now by Stirling's approximation, we have $\log n! \sim n \log n - n$.
$\therefore k_0 \in \Omega(n \log n)$ □

3 References

1. Introduction To Algorithms by Thomas H Cormen,Carles E Leiserson,Ronald L Rivest,Clifford Stein