Asymptotic Notations
Recurrence

SWE2016-44

Performance analysis

- **Given two algorithms for a task, how do we find out which one is better?**
  - **Naïve way:** implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.
    1. It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
    2. It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.
  - **Asymptotic Analysis**: the big idea that handles above issues in analyzing algorithms. We evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.
- Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms.

Worst, Average and Best Cases

- We can have three cases to analyze an algorithm:
  1. **Worst Case**: the case that causes maximum number of operations to be executed.
  2. **Average Case**: Sum all the calculated values and divide the sum by total number of inputs.
  3. **Best Case**: the case that causes minimum number of operations to be executed.

- Examples
  1. **Merge sort**: $\Theta(n \ln n)$ operations in all cases
  2. **Insertion sort**: the worst case occurs when the array is reverse sorted and the best case occurs when the array is sorted in the same order as output.

Asymptotic Notations

- **Asymptotic analysis**: a measure of efficiency of algorithms that doesn’t depend on machine specific constants, and doesn’t require algorithms to be implemented and time taken by programs to be compared.

- **Asymptotic notations**: mathematical tools to represent time complexity of algorithms for asymptotic analysis.

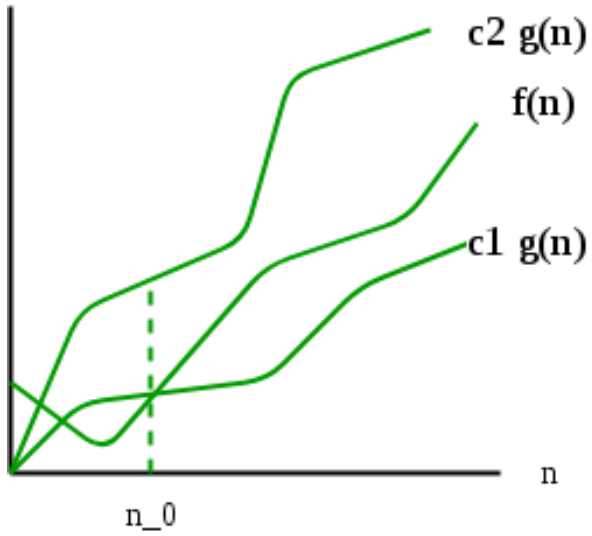
- **Types of asymptotic notations**
  1. **Θ** notation
  2. **Big O** notation
  3. **Big Ω** notation

Θ Notation

- **Θ Notation**: The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.
  - A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.

$$3n^3 + 6n^2 + 6000 = \boldsymbol{\Theta}(n^3)$$

- For a given function $g(n)$, we denote $\Theta\big(g(n)\big)$ is following set of functions.



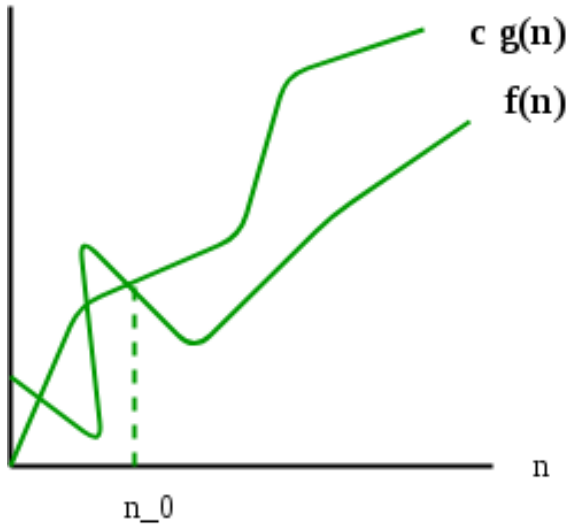
f(n) = theta(g(n))

$\Theta\big(g(n)\big) = \{\, f(n)$: there exist positive constants c_1, c_2 and n_0 such that

$0 \le c_1 * g(n) \le f(n) \le c_2 * g(n)$ for all $n \ge n_0\}$

Big O Notation

- **Big O Notation**: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.
  - For example, consider the case of Insertion Sort. It takes linear time ($O(n)$) in best case and quadratic time ($O(n^2)$) in worst case. We can safely say that the time complexity of Insertion sort is $O(n^2)$. Note that $O(n^2)$ also covers linear time.



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

- For a given function $g(n)$, we denote $O\big(g(n)\big)$ is following set of functions.

$O\big(g(n)\big) = \{\, f(n)$: there exist positive constants c and n_0 such that

$0 \le f(n) \le c^{*}g(n)$ for all $n \ge n_0\}$

Big O Notation (more)

- The general step wise procedure for Big-O runtime analysis is as follows:
  1. Figure out what the input is and what n represents.
  2. Express the maximum number of operations, the algorithm performs in terms of n.
  3. Eliminate all excluding the highest order terms.
  4. Remove all the constant factors.

- *Constant Multiplication:*

If $f(n) = c.g(n)$, then $O(f(n)) = O(g(n))$; where c is a nonzero constant.

- *Polynomial Function:*

If $f(n) = a_0 + a_1.n + a_2.n^2 + \text{—-} + a_m.n^m$, then $O(f(n)) = O(n^m)$.

- *Summation Function:*

If $f(n) = f_1(n) + f_2(n) + \text{—-} + f_m(n)$ and $f_i(n) \leq f_{i+1}(n)$ $\forall$ i=1, 2, —-, m,

then $O(f(n)) = O(\max(f_1(n), f_2(n), \text{—-}, f_m(n)))$.

- *Logarithmic Function:*

If $f(n) = \log_a n$ and $g(n)=\log_b n$, then $O(f(n))=O(g(n))$

; all log functions grow in the same manner in terms of Big-O.

Big O Notation (more)

- *A logarithmic algorithm – O(logn)*

Runtime grows logarithmically in proportion to n.

- *A linear algorithm – O(n)*

Runtime grows directly in proportion to n.

- *A superlinear algorithm – O(nlogn)*

Runtime grows in proportion to n.

- *A polynomial algorithm – $O(n^c)$*

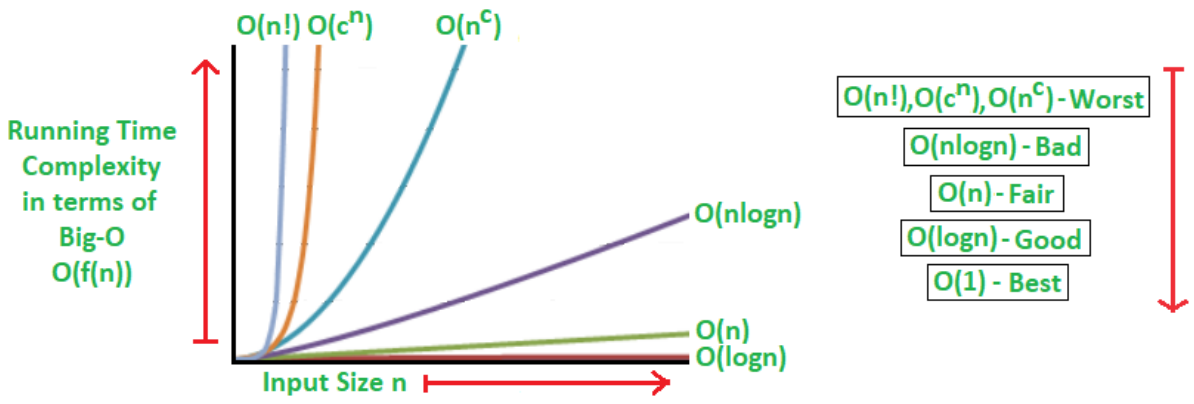
Runtime grows quicker than previous all based on n.

- *A exponential algorithm – $O(c^n)$*

Runtime grows even faster than polynomial algorithm based on n.

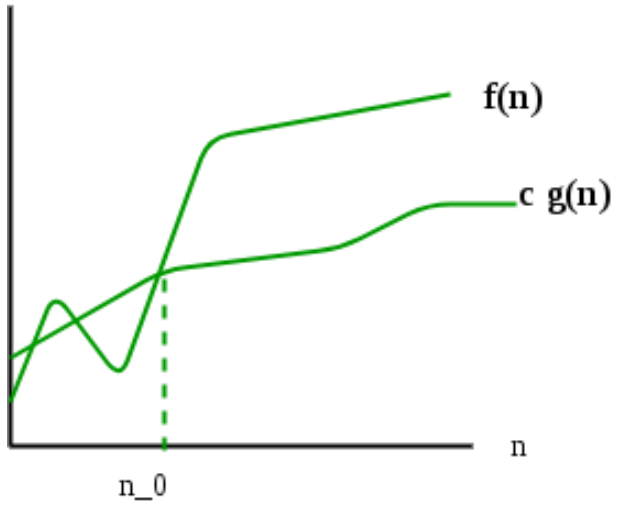
- *A factorial algorithm – O(n!)*

Runtime grows the fastest and becomes quickly unusable for even small values of n.



Big Ω Notation

- **Big Ω Notation**: Just as Big O notation provides an asymptotic upper bound on a function, **Ω** notation provides an asymptotic lower bound.
  - the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.



f(n) = Omega(g(n))

- For a given function $g(n)$, we denote $\mathbf{\Omega}\big(g(n)\big)$ is following set of functions.

$\mathbf{\Omega}\big(g(n)\big) = \{\, f(n)$: there exist positive constants c and n_0 such that

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

Examples

- **Insertion Sort**
  - Θ notation
    - Best Case: $T(n) = 5n - 5$ → $\Theta(n)$
    - Worst Case: $T(n) = \frac{3}{2}n^2 + \frac{1}{2}n - 2$ → $\Theta(n^2)$
    - Average Case: $T(n) = \frac{3}{4}n^2 + \frac{11}{4}n - \frac{7}{2}$ → $\Theta(n^2)$
  - O notation: $O(n^2)$
  - Ω notation: $\Omega(n^2)$

- **Merge two sorted arrays**
  - Θ notation
    - Best Case: $T(n) = 4m + n + 2$ → $\Theta(m + n)$
    - Worst Case: $T(n) = 4m + 4n - 1$ → $\Theta(m + n)$
    - Average Case: $T(n) = 4m + \frac{5}{2}n + \frac{1}{2}$ → $\Theta(m + n)$
  - O notation: $O(m + n)$
  - Ω notation: $\Omega(m + n)$

Analysis of Loops

- **O(1)**: Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

```
// set of non-recursive and non-loop statements
```

```
// Here c is a constant
for (int i = 1; i <= c; i++) {
     // some O(1) expressions
}
```

- **O(n)**: Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example, following functions have O(n) time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
     // some O(1) expressions
}

for (int i = n; i > 0; i -= c) {
     // some O(1) expressions
}
```

Analysis of Loops

- **$O(n^c)$**: Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example, the following sample loops have $O(n^2)$ time complexity.

```
for (int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
       // some O(1) expressions
    }
}

for (int i = n; i > 0; i -= c) {
    for (int j = i+1; j <=n; j += c) {
       // some O(1) expressions
}
```

- **$O(\log n)$**: Time Complexity of a loop is considered as O(log n) if the loop variables is divided / multiplied by a constant amount.

```
for (int i = 1; i <=n; i *= c) {
    // some O(1) expressions
}
for (int i = n; i > 0; i /= c) {
    // some O(1) expressions
}
```

Analysis of Loops

- **O(log log n)**: Time Complexity of a loop is considered as O(log log n) if the loop variables is reduced / increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {
    // some O(1) expressions
}
//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 1; i = fun(i)) {
    // some O(1) expressions
}
```

- **Time complexities of consecutive loops**: When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

```
for (int i = 1; i <=m; i += c) {
     // some O(1) expressions
}
for (int i = 1; i <=n; i += c) {
     // some O(1) expressions
}
Time complexity of above code is O(m) + O(n) which is O(m+n)
If m == n, the time complexity becomes O(2n) which is O(n).
```

Solving Recurrences

- When we analyze recursive algorithms, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes.
  - For example, in Merge Sort, to sort a given array,
    - We divide it in two halves and recursively repeat the process for the two halves.
    - Finally we merge the results.
    - Time complexity of Merge Sort: $T(n) = 2T\left(\frac{n}{2}\right) + cn$.
  - There are many other algorithms like Binary Search, Tower of Hanoi, etc.

- There are mainly three ways for solving recurrences.
  1. Substitution Method
  2. Recurrence Tree Method
  3. Master Method

Substitution Method

- We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

```
For example consider the recurrence T(n) = 2T(n/2) + n

We guess the solution as T(n) = O(nLogn). Now we use induction
to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it is true
for values smaller than n.

T(n) = 2T(n/2) + n
    <= cn/2Log(n/2) + n
    =  cnLogn - cnLog2 + n
    =  cnLogn - cn + n
    <= cnLogn
```

Recurrence Tree Method

- In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

```
For example consider the recurrence relation
T(n) = T(n/4) + T(n/2) + cn²

           cn²
         /     \
     T(n/4)     T(n/2)

If we further break down the expression T(n/4) and T(n/2),
we get following recursion tree.

                cn²
           /           \
       c(n²)/16      c(n²)/4
      /      \          /     \
  T(n/16)     T(n/8)  T(n/8)    T(n/4)
```

```
Breaking down further gives us following
                 cn²
            /            \
       c(n²)/16          c(n²)/4
       /      \            /      \
c(n²)/256   c(n²)/64  c(n²)/64    c(n²)/16
 /    \      /    \    /    \       /    \

To know the value of T(n), we need to calculate sum of tree
nodes level by level. If we sum the above tree level by level,
we get the following series
T(n)  = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ....
The above series is geometrical progression with ratio 5/16.

To get an upper bound, we can sum the infinite series.
We get the sum as (n²)/(1 - 5/16) which is O(n²)
```

Master Method

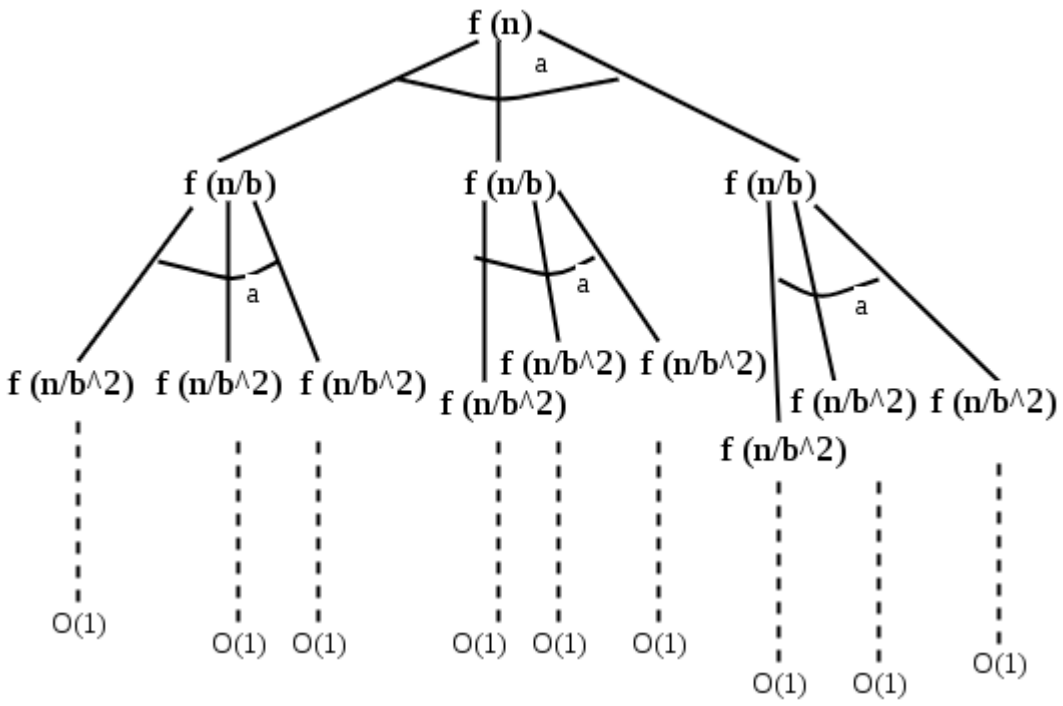
- Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ where } a \geq 1 \text{ and } b > 1$$

- There are following three cases:
  1. If $f(n) = \Theta(n^c)$ where $c < log_b a$ then $T(n) = \Theta\left(n^{log_b a}\right)$
  2. If $f(n) = \Theta(n^c)$ where $c = log_b a$ then $T(n) = \Theta(n^c \, log \, n)$
  3. If $f(n) = \Theta(n^c)$ where $c > log_b a$ then $T(n) = \Theta\left(f(n)\right)$

Master Method

- Master method is mainly derived from recurrence tree method. If we draw recurrence tree of $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, we can see that the work done at root is $f(n)$ and work done at all leaves is $\Theta(n^c)$ where c is $log_b\, a$. And the height of recurrence tree is $log_b\, n$.



In recurrence tree method, we calculate total work done.

- If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1).
- If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2).
- If work done at root is asymptotically more, then our result becomes work done at root (Case 3).

Master Method

- **Notes**
  - It is not necessary that a recurrence of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence $T(n) = aT\left(\frac{n}{b}\right) + n\,log\,n$ cannot be solved using master method.
  - Case 2 can be extended for $f(n) = \Theta(n^c\ log^k n)$. If $f(n) = \Theta(n^c\ log^k n)$ for some constant $k \geq 0$ and $c = log_b\,a$, then $T(n) = \Theta\left(n^c\ log^{k+1} n\right)$

- **Examples**
  - Merge Sort: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$. It falls in case 2 as c is 1 and $log_b\,a$ is also 1. So the solution is $\Theta(n\ log\ n)$.
  - Binary Search: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1)$. It also falls in case 2 as c is 0 and $log_b\,a$ is also 0. So the solution is $\Theta(log\ n)$.