Lecture 2

Asymptotic Analysis

Md. Asif Bin Khaled

Ram Model

Machine-Independent algorithm design upon a hypothetical computer called the **Random Access Machine(RAM)**. Under this model of computation we are confronted with a computer where:

1. Each simple operation(+,*,-,=,if,call) takes exactly one step.
2. Loops and subroutines are not considered simple operations. Instead, they are the composition of many single-step operations.
3. The data types in the model are integer and floating point.

RAM Model Continued.

---

4. Each memory access takes exactly one step. However, we have as much as memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk.
5. Runtime is measured by counting up the number of steps an algorithm takes on a given problem instance. For example, multiplying or adding two numbers take only one step. However in reality multiplying is more costly.

Despite being an impractical model, the RAM model helps us to analyze algorithms in a machine-independent way.

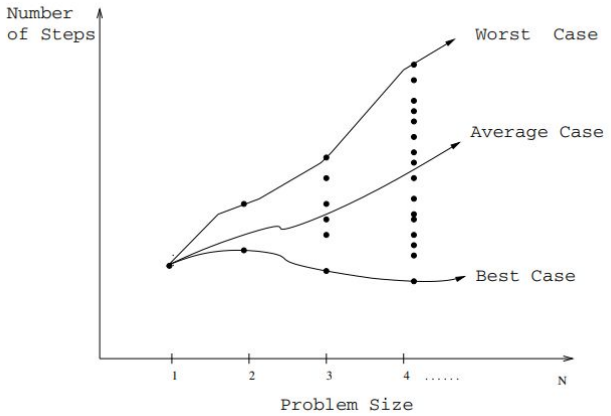
Best, Worst & Average Case Complexity

Using RAM model of computation, we can count how many steps our algorithm takes on any given input instance by executing it however, to understand how good or bad an algorithm is in general, we must know how it works over all instance. For example, in the problem of sorting, the set of possible input instances consists of all possible arrangements of n keys, over all possible values of n.

The running time of an algorithm on a particular input is the **number of primitive operations or steps executed.**

Best, Worst & Average Case Complexity Continued.

We can represent each input instance as a point on a graph (for example the problem of sorting) where the x-axis represents the size of the input problem and the y-axis denotes the number of steps taken by the algorithm in this instance.



Best, Worst & Average Case Complexity Continued.

We can define three functions from all these points,

1. **Worst Case Complexity**: The worst case complexity of the algorithm is the function defined by the maximum number of steps taken in any instance of size n.
2. **Best Case Complexity**: The best case complexity of the algorithm is the function defined by the minimum number of steps taken in any instance of size n.
3. **Average Case Complexity**: The average case complexity of the algorithm, which is the function defined by the average number of steps over all instances of size n.

Best, Worst & Average Case Complexity Continued.

The important thing is to realize that each of these time complexities define **numerical function**, representing time vs problem size. These functions are as well defined as any other numerical function, be it $y = x^3 + 3$ or

$$y = 6541n^3 + 3213n^2 + 56n + 231$$

However time complexities are such complicated function that we must simplify them to work with them. To resolve this we will be introducing **notations**.

Loop Invariant

A loop invariant is a property of a program loop that is true **before** and **after** each iteration.

```
Max(A,n) {
      // m equals the maximum value in a[1...1]
      m = A[1];
      for i=2 to n:
            // m equals the maximum value in A[1...i-1]
            if (m < A[i])
                  m = A[i];
            // m equals the maximum value in A[1...i]
      // m equals the maximum value in A[1...n]
      return m;
}
```

Loop Invariant Continued

The loop invariant must be true

1. Before the loop starts.
2. Before each iteration of the loop.
3. After the loop terminates.

The Problem of Sorting

If we want to solve a problem then we would like to have the properly define the problem otherwise there may arise a lot of ambiguity.

Input: Sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: Permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

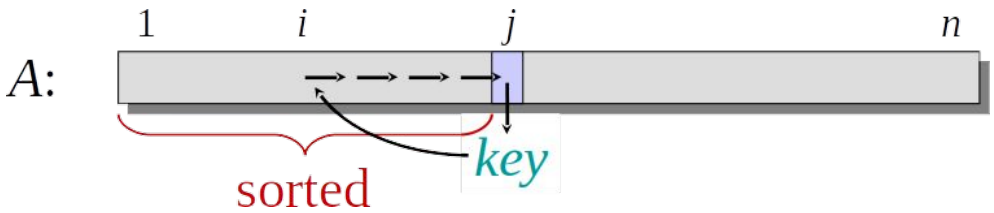
For example,

Input: 12, 7, 88, 5, 46

Output: 5, 7, 12, 46, 88

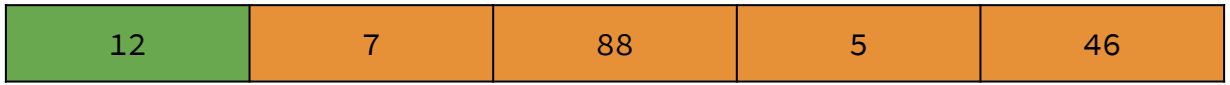
Insertion Sort Pseudocode

```
Insertion_Sort(A):
    for j=2 to A.length:
        key=A[j]
        //insert A[j] into the sorted sequence A[1...j-1]
        i=j-1
        while i>0 and A[i]>key:
            A[i+1]=A[i]
            i=i-1
        A[i+1]=key
```



Simulation of Insertion Sort

A single element is always sorted.

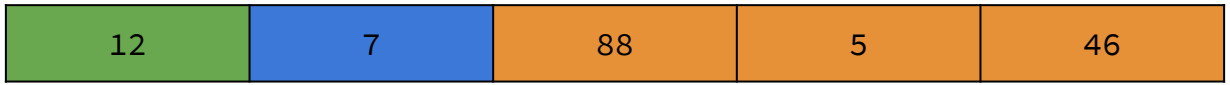


Simulation of Insertion Sort

j=2

i=j-1

key=7

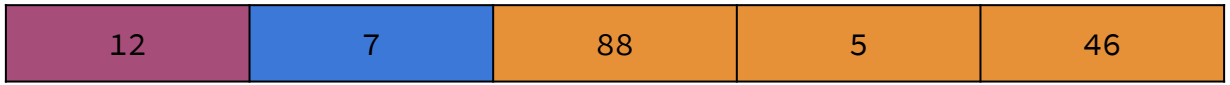


Simulation of Insertion Sort

j=2

i=1

key=7



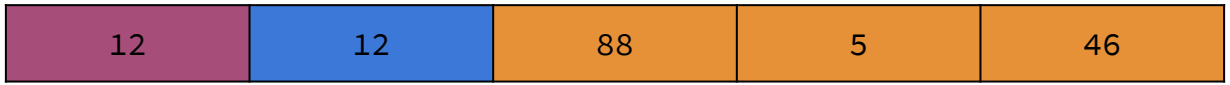
Simulation of Insertion Sort

Comparing A[i]>key and if true shifting, A[i+1]=A[i]

j=2

i=1

key=7



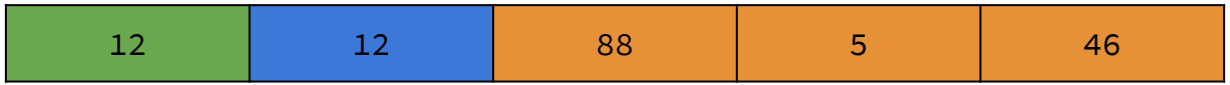
Simulation of Insertion Sort

End of the inner loop

j=2

i=0

key=7



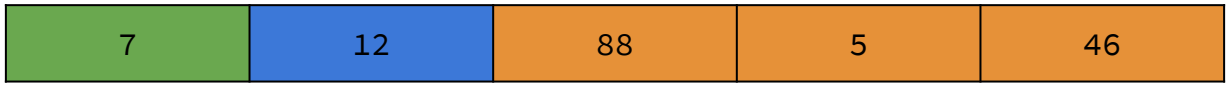
Simulation of Insertion Sort

Assigning A[i+1]=key

j=2

i=0

key=7



Simulation of Insertion Sort

```
j=3
i=j-1
key=88
```

Simulation of Insertion Sort

j=3

i=2

key=88

Simulation of Insertion Sort

Comparing A[i]>key and if true shifting, A[i+1]=A[i]

j=3

i=2

key=88

Simulation of Insertion Sort

```
End of the inner loop
j=3
i=2
key=88
```

Simulation of Insertion Sort

Assigning A[i+1]=key

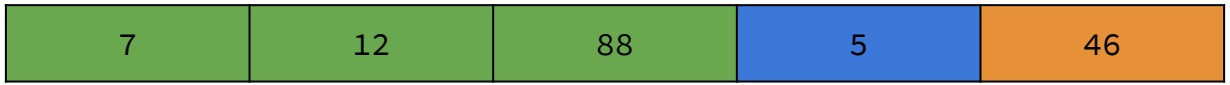
j=3

i=2

key=88

Simulation of Insertion Sort

```
j=4
i=j-1
key=5
```



Simulation of Insertion Sort

j=4

i=3

key=5

Simulation of Insertion Sort

Comparing A[i]>key and if true shifting, A[i+1]=A[i]

j=4

i=3

key=5

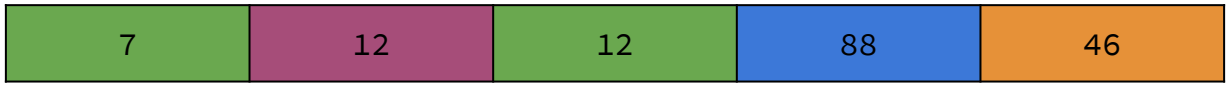
Simulation of Insertion Sort

Comparing A[i]>key and if true shifting, A[i+1]=A[i]

j=4

i=2

key=5



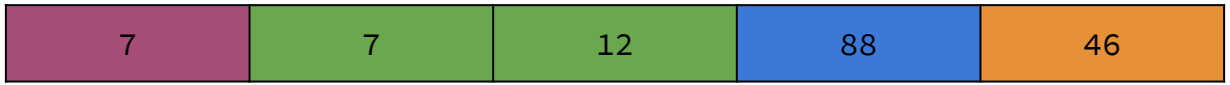
Simulation of Insertion Sort

Comparing A[i]>key and if true shifting, A[i+1]=A[i]

j=4

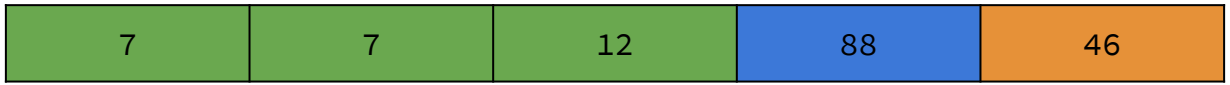
i=1

key=5



Simulation of Insertion Sort

```
End of the inner loop
j=4
i=0
key=5
```



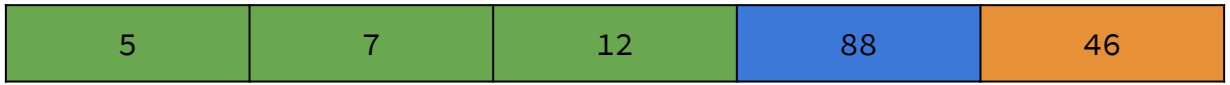
Simulation of Insertion Sort

Assigning A[i+1]=key

j=4

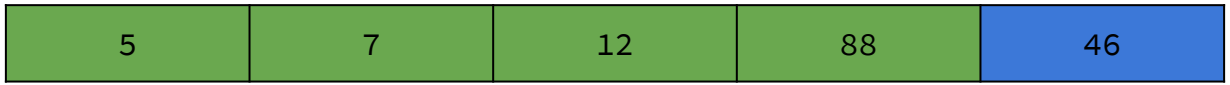
i=0

key=5



Simulation of Insertion Sort

```
j=5
i=j-1
key=46
```

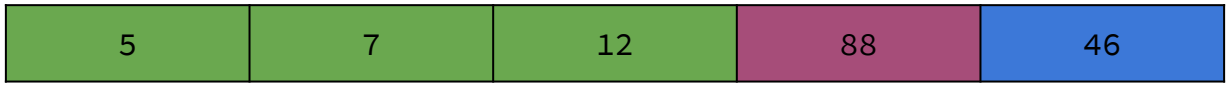


Simulation of Insertion Sort

j=5

i=4

key=46



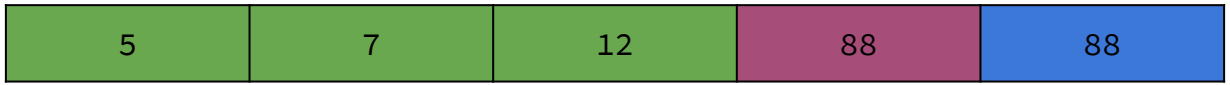
Simulation of Insertion Sort

Comparing A[i]>key and if true shifting, A[i+1]=A[i]

j=5

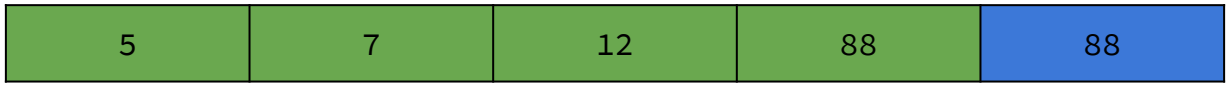
i=4

key=46



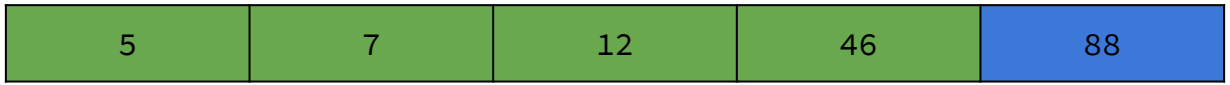
Simulation of Insertion Sort

```
End of the inner loop
j=5
i=3
key=46
```



Simulation of Insertion Sort

```
Assigning A[i+1]=key
j=5
i=3
key=46
```



Analysis of Insertion Sort

---

1. The running time of insertion sort depends on the input. An already sorted sequence is easier to sort.
2. We can parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
3. Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

Analysis of Insertion Sort Continued.

Let us assume for now that a constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another, but we shall assume that each execution of the i^{th} line takes c_i, where c_i is a constant. This viewpoint is in keeping with the RAM model.

Analysis of Insertion Sort Continued.

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Analysis of Insertion Sort (Best Case) Continued.

Best case occurs if the array is already sorted. We can express this running time as **an + b** for constants a and b that depend on the statement costs c_i. It is thus a linear function of n. [**Ω(n),** yet to be discussed.]

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$T(n) = c_1 n + c_2 n - c_2 + c_4 n - c_4 + c_5 n - c_5 + c_8 n - c_8$$
$$T(n) = c_1 n + c_2 n + c_4 n + c_5 n + c_8 n - c_2 - c_4 - c_5 - c_8$$
$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$
$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n + (-(c_2 + c_4 + c_5 + c_8))$$
$$T(n) = an + b$$

Analysis of Insertion Sort (Worst Case) Continued.

When the array is in reverse sorted order which means that the array is in decreasing order, is the worst case for this algorithm. We can express this worst case running time as **$an^2 + bn + c$** for constants a, b and c that again depend on the statement costs c_i. It is thus a quadratic function of n. [**$O(n^2)$**, yet to be discussed.]

$$S = 1 + 2 + 3 + 4 + ... + (n-1) \cdots\cdots\cdots\cdots (i)$$
$$S = (n-1) + (n-2) + (n-3) + (n-4) + ... + 1 \cdots\cdots\cdots\cdots (ii)$$

Adding equation (i) and (ii)

$$2S = n + n + n + ... + n$$
$$2S = (n-1)n$$
$$S = \frac{n(n-1)}{2}$$

Analysis of Insertion Sort (Worst Case) Continued.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 (\frac{n(n+1)}{2} - 1) + c_6 (\frac{n(n-1)}{2}) + c_7 (\frac{n(n-1)}{2}) + c_8(n-1)$$

$$T(n) = c_1 n + c_2 n - c_2 + c_4 n - c_4 + c_5 \frac{n^2}{2} + c_5 \frac{n}{2} - c_5 + c_6 \frac{n^2}{2} - c_6 \frac{n}{2}) + c_7 \frac{n^2}{2} - c_7 \frac{n}{2}) + c_8 n - c_8$$

$$T(n) = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}) n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8) n - (c_2 + c_4 + c_5 + c_8)$$

$$T(n) = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}) n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8) n + (-(c_2 + c_4 + c_5 + c_8))$$

$$T(n) = an^2 + bn + c$$

Analysis of Insertion Sort (Average Case) Continued.

So, how can we express the following?

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} \frac{t_j}{2} + c_6 \sum_{j=2}^{n} (\frac{t_j}{2} - 1) + c_7 \sum_{j=2}^{n} (\frac{t_j}{2} - 1) + c_8(n-1)$$

Machine-Independent Time

---

1. **Relative speed (on the same machine)**: Whether an algorithm is better when comparing by running on the same machine.
2. **Absolute speed (on different machines)**: Whether an algorithm is better no matter what machine it is run on.

The main idea of the asymptotic analysis is to ignore the machine-dependent constants and look at growth of **T(n)** as **n → ∞** (n approaches infinity).

It is convenient to define the notion of **steps** with **running time** so that it is as machine-independent as possible.

Asymptotic Analysis

When analyzing the running time or space usage of programs, we usually try to estimate the time or space as function of the input size. For example, when analyzing the worst case running time of a function that sorts a list of numbers, we will be concerned with how long it takes as a function of the length of the input list.

In mathematical analysis, asymptotic analysis, also known as asymptotics, is a method of describing limiting behavior.

For large inputs, the slower the asymptotic growth rate, the better the algorithm is.

Asymptotic Analysis Continued

The asymptotic behavior of a function f(n) (such as f(n)=c⋆n or f(n)=c⋆n^2, etc.) refers to the growth of f(n) as n gets large.

Suppose that we are interested in the properties of a function f(n) as n becomes very large. If f(n) = n^2 + 3n, then as n becomes very large, the term 3n becomes insignificant compared to n^2. The function f(n) is said to be "asymptotically equivalent to n^2, as n → ∞". This is often written symbolically as f(n) ~ n^2, which is read as "f(n) is asymptotic to n^2".

Asymptotic Function Order

Function	Comment
n^0 or 1	Constant
lgn	logarithmic
√n	
n	linear
nlgn	
n^k	Polynomial, k>1
a^n	Exponential,

Increasing

Asymptotic Notation

The notation we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers N={0,1,2,....}. Such notations are convenient for describing the worst case running time function T(n), which usually is defined only on integer input sizes.

Big O (Oh) Notation (Upper Bounds)

We write **f(n) = O(g(n))** if there exist constants **c > 0, n_0 > 0** such that,

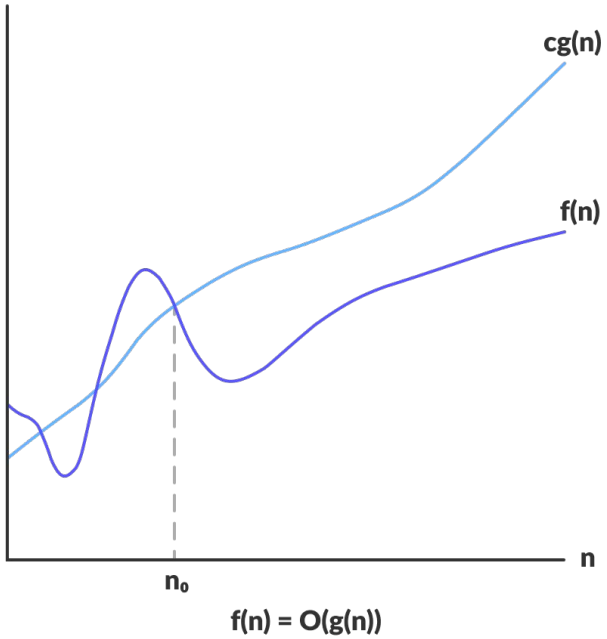
$$0 \leq f(n) \leq c*g(n)$$

for all **n** ≥ **n_0**.

Set Definition,

O(g(n)) = {**f(n)**: there exist constants **c > 0, n_0 > 0** such that **0 ≤ f(n) ≤ c*g(n)** for all **n ≥ n_0**}

Big O (Oh) Notation (Upper Bounds) Continued



Big O (Oh) Notation (Upper Bounds) Continued

When we say that an algorithm runs in time T(n), we mean that T(n) is an upper bound on the running time that holds for all inputs of size n. This is called worst-case analysis. The algorithm may very well take less time on some inputs of size n, but it doesn't matter.

If an algorithm takes $T(n)=c*n^2+k$ steps on only a single input of each size n and only n steps on the rest, we still say that it is a quadratic algorithm.

Big O (Oh) Notation (Upper Bounds) Continued.

We use O-notation to give an upper bound on a function, within a constant factor. We write f(n)=O(g(n)) to indicate that a function f(n) is a member of the set O(g(n)). When we are writing f(n)=O(g(n)), we are claiming that some constant multiple of g(n) is an asymptotic upper bound on f(n), with no claim about how tight an upper bound it is.

$2n^2 = O(n^3)$ is a one way equality. In set definition it is similar to $2n^2 \in O(n^3)$.

Big O (Oh) Notation (Upper Bounds) Proof

---

1. $2n^2 = O(n^3)$

   here,

   $f(n)=n^2$, $g(n)=n^3$

   hence,

   $2n^2 \leq c*n^3$

   $2n^2/n^2 \leq c*n^3/n^2$

   $2 \leq c*n$

   $c = 1$ & $n_0 = 2$

2. $10n^3 = O(n^3)$

   here,

   $f(n)=10n^3$, $g(n)=n^3$

   hence,

   $10n^3 \leq c*n^3$

   $10n^3/n^3 \leq c*n^3/n^3$

   $10 \leq c$

   $c = 10$ & $n_0 = 1$

Big O (Oh) Notation (Upper Bounds) Proof Continued.

---

3. $3n + 2 = O(n)$

here,

$f(n)=3n + 2$, $g(n)=n$

hence,

$3n + 2 \leq c*n$

$3n/n + 2/n \leq c*n/n$

$3 + 2/n \leq c$

$c = 5$ & $n_0 = 1$

4. $10n^4 = O(n^3)$

here,

$f(n)=10n^4$, $g(n)=n^3$

hence,

$10n^4 \leq c*n^3$

$10n^4/n^3 \leq c*n^3/n^3$

$10n \leq c$

As c can not ever outrun n hence, the inequality does not stand.

Big Ω (Omega) Notation (Lower Bounds)

We write **f(n) = Ω(g(n))** if there exist constants **c > 0, n_0 > 0** such that,

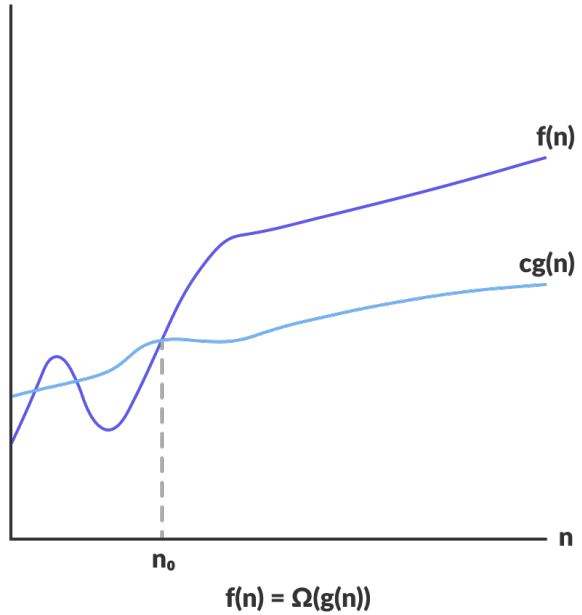
$$0 \leq c*g(n) \leq f(n)$$

for all **n ≥ n_0**.

Set Definition,

Ω(g(n)) = {**f(n)**: there exist constants **c > 0, n_0 > 0** such that **0 ≤ c*g(n) ≤ f(n)** for all **n ≥ n_0**}

Big Ω (Omega) Notation (Lower Bounds) Continued



Big Θ (theta) Notation (Tight Bounds)

We write $f(n) = \Theta(g(n))$ if there exist constants $c_1 > 0$, $c_2 > 0$, $n_0 > 0$ such that,

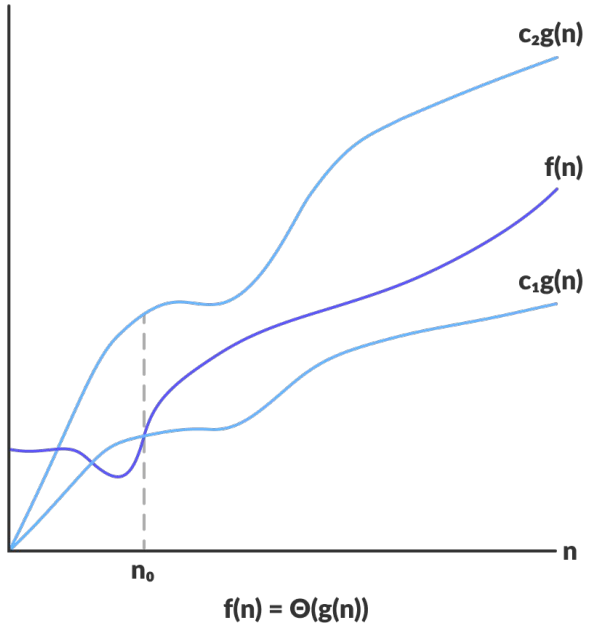
$$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$$

for all $n \geq n_0$.

Set Definition,

$\Theta(g(n))$ = {$f(n)$: there exist constants $c_1 > 0$, $c_2 > 0$, $n_0 > 0$ such that $0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0$}

Big Θ (theta) Notation (Tight Bounds) Continued



Big Θ (theta) Notation (Tight Bounds) Continued

$c_1 * g(n) \leq f(n)$ is $\Omega(g(n))$

$f(n) \leq c_2 * g(n)$ is $O(g(n))$

Hence,

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

is

$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

Big Θ (theta) Notation (Tight Bounds) Continued

In the average case we do not bound the worst case running time, but try to calculate the expected time spent on a randomly chosen input. This kind of analysis is generally harder, since it involves probabilistic arguments and often requires assumptions about the distribution of inputs that may be difficult to justify.

The average case of an algorithm can be more useful because sometimes the worst case behavior of an algorithm is misleadingly bad.

Complexity Examples

1.
```
int a,b,c;
for(int i=0; i<n; i++){
     a=b;
     b=c;
}
```

Complexity: **O(n)**

2.
```
int a,b,c;
for(int i=0; i<n; i+=2){
     a=b;
     b=c;
}
```

Complexity: **O(n)**

Complexity Examples Continued

3.
```
int a,b,c;
for(int i=1; i<n; i*=2){
     a=b;
     b=c;
}
```

Complexity: $\mathbf{O(lg_2 n)}$

4.
```
int a,b,c;
for(int i=1; i<n; i*=3){
     a=b;
     b=c;
}
```

Complexity: $\mathbf{O(lg_3 n)}$

Complexity Examples Continued

5.
```
int a,b,c;
for(int i=n; i>=0; i/=2){
     a=b;
     b=c;
}
```

Complexity: $\mathbf{O(lg_2 n)}$

6.
```
int a,b,c;
for(int i=n; i>=0; i/=3){
     a=b;
     b=c;
}
```

Complexity: $\mathbf{O(lg_3 n)}$

Complexity Examples Continued

7.
```
int i=0,s=0;
while(s<=n){
     i++;
     s=s+i;
}
```

Complexity: **O(√n)**

8.
```
int a,b,c;
for(int i=n; i>=0; i/=3){
     a=b;
     b=c;
}
for(int i=n; i>=0; i/=2){
     a=b;
     b=c;
}
```

Complexity: **$O(lg_3 n + lg_2 n) \rightarrow O(lg_2 n)$**

Complexity Examples Continued

9.
```
int
for(int i=0; i<m; i++){
     for(int j=0; i<n; j++){
          a=b;
          b=c;
     }
}
```

Complexity: **O(mn)**

```
a,b,c;
```

10.
```
int a,b,c;
for(int i=0; i<p; i++){
     for(int j=0; j<q; j++){
          a=b;
     }
     for(int j=0; j<r; j++){
          a=b;
     }
}
```

Complexity: **O(p(q+r))**

Complexity Examples Continued

11.
```
int a = 0;
for (i = 0; i < n; i++) {
    for (j = n; j > i; j--) {
        a = a + i + j;
    }
}
```

Complexity: **$O(n^2)$**

12.
```
int a = 0;
while (n > 0) {
    a += n;
    n /= 2;
}
```

Complexity: **$O(\lg_2 n)$**

Shortcomings of Asymptotic Analysis

In practice, other considerations beside asymptotic analysis are important when choosing between algorithms. Let algorithm A be asymptotically better than algorithm B,

1. **Implementation Complexity:** Algorithms with better complexity are often (much) more complicated. This can increase coding time and the constants.
2. **Small Input Sizes** Asymptotic analysis ignores small input sizes. At small input sizes, constant factors or low order terms could dominate running time, causing B to outperform A.

Shortcomings of Asymptotic Analysis Continued

---

3. **Worst Case versus Average Case Performance:** If A has better worst case performance than B, but the average performance of B given the expected input is better, then B could be a better choice than A. Conversely, if the worst case performance of B is unacceptable (say for life-threatening or mission-critical reasons), A must still be used.

Reference

---

1. https://www.cs.cornell.edu/courses/cs312/2004fa/lectures/lecture16.htm
2. https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation