Data Structures

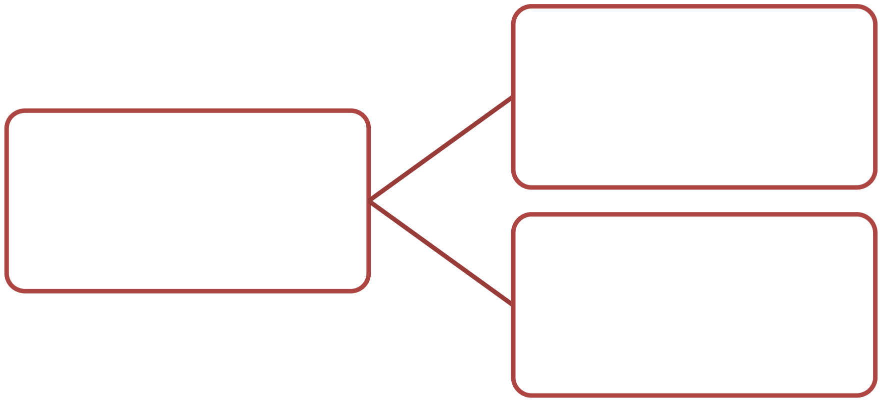
UCS301

Asymptotic Notations

Department of CSE
Thapar Institute of Engineering and Technology, Patiala

Algorithm and its complexity

- It is the sequence of instruction which should be executed to perform meaningful task.
- Efficiency or complexity of an algorithm is analyzed in terms of
  - cpu time and
  - memory.



- Amount of computational time required by an algorithm to perform complete task.
- Amount of memory required by an algorithm to complete its execution.

3 cases to analyze an algorithm

1) *Worst Case Analysis* (Usually Done) –
   – we calculate upper bound on running time of an algorithm.
   – We must know the case that causes maximum number of operations to be executed.

2) *Best Case Analysis* (Bogus)
   - we calculate lower bound on running time of an algorithm.
   - We must know the case that causes minimum number of operations to be
   executed.

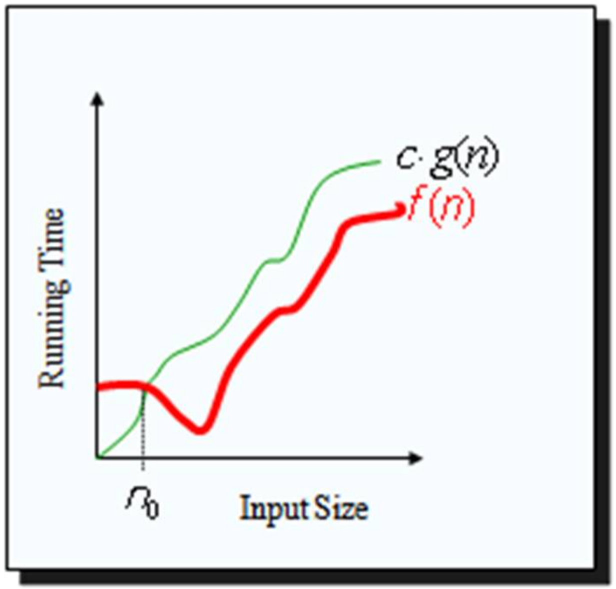
3) *Average Case Analysis* (Sometimes done)
   - we take all possible inputs and calculate computing time for all of the inputs.
   - Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases.

Asymptotic Notation

- They are the mathematical tools to represent the time complexity of algorithms for asymptotic analysis.
- Evaluate the performance of an algorithm in terms of input size
- The notation we use to describe the asymptotic running time of an algorithm are defined in terms of function whose domain are the set of natural number N = {0,1,2,...}
- It also describe the behaviour of time or space complexity for large instance characteristics.
- Calculate, how the time (or space) taken by an algorithm increases with the input size
- There are three Asymptotic Notations
  - Big-oh (O)   Omega(Ω)   Theta(Θ)

Asymptotic Notation: Big-oh Notation (O)

- Asymptotic upper bound used for worst-case analysis
- Let f(n) and g(n) are functions over non-negative integers
- if there exists constants c and n_0, such that
- $f(n) \le c\, g(n)$ for $n \ge n_0$
- Then we can write $f(n) = O(g(n))$
- Example $f(n) = 2n + 1$, $g(n) = 3n$
- i.e $f(n) \le 3n$
- so we can say $f(n) = O(n)$ when $c = 3$



Example: Big-oh Notation (O)

- Consider $f(n) = 2n+2$, $g(n) = n^2$.
- Find some constant c such that $f(n) \leq g(n)$
- For n=1, $f(n) = 2*1+2 = 4$, $g(n) = 1$ - > $f(n)>g(n)$
- For n=2, $f(n) = 2*2+2 = 6$, $g(n) = 4$ -> $f(n)>g(n)$
- For n=3, $f(n) = 2*3+2 = 8$, $g(n) = 9$ -> $f(n)<g(n)$

- Thus for $n > 2$, $f(n) < g(n)$ ->always an upper bound.

Example: Big-Oh

Show that: $\mathbf{n^2/2 - 3n = O(n^2)}$

- Determine positive constants $\mathbf{c_1}$ and $\mathbf{n_0}$ such that

$$n^2/2 - 3n \leq c_1 n^2 \text{ for all } n \geq n_0$$

- Diving by $\mathbf{n^2}$

$$1/2 - 3/n \leq c_1$$

- For:

  $\mathbf{n = 1}$, $\quad \mathbf{1/2 - 3/1 \leq c_1}$ (Holds for $\mathbf{c_1 \geq 1/2}$)

  $\mathbf{n = 2}$, $\quad \mathbf{1/2 - 3/2 \leq c_1}$ (Holds and so on…)

- The inequality holds for any $\mathbf{n \geq 1}$ and $\mathbf{c_1 \geq 1/2}$.
- Thus by choosing the constant $\mathbf{c_1 = 1/2}$ and $\mathbf{n_0 = 1}$, one can verify that $\mathbf{n^2/2 - 3n = O(n^2)}$ holds.

Asymptotic Notation: Omega Notation (Ω)

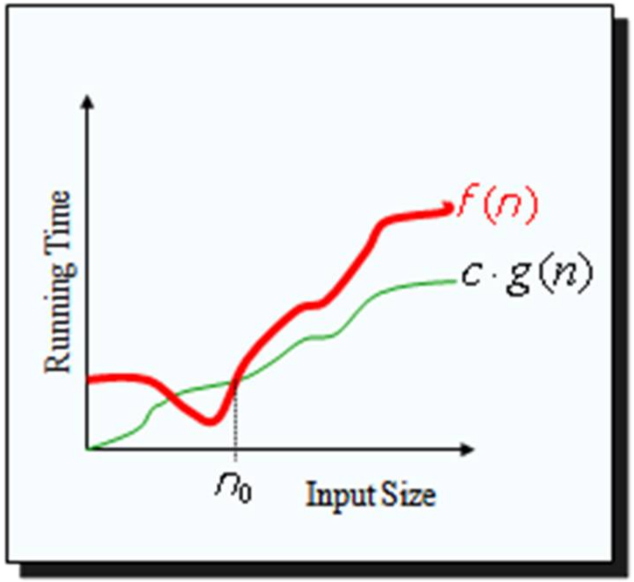
- Asymptotic lower bound used to describe best-case running times
- Let f(n) and g(n) are functions over non-negative integers
- if there exists constants c and n_0, such that

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

- Then we can write $f(n) = \Omega(g(n))$,
- Example $f(n) = 18n+9$, $g(n) = 18n$

  i.e $f(n) \geq 18n$

so we can say $f(n) = \Omega(n)$ when $c = 18$



Example: Big-Omega

Show that: $n^2/2 - 3n = \Omega(n^2)$

- Determine positive constants c_1 and n_0 such that

$$c_1 n^2 \le n^2/2 - 3n \quad \text{for all } n \ge n_0$$

- Diving by n^2

$$c_1 \le 1/2 - 3/n$$

- For: $n = 1$, $c_1 \le 1/2 - 3/1$ (Not Holds)
  $n = 2$, $c_1 \le 1/2 - 3/2$ (Not Holds)
  $n = 3$, $c_1 \le 1/2 - 3/3$ (Not Holds)
  $n = 4$, $c_1 \le 1/2 - 3/4$ (Not Holds)
  $n = 5$, $c_1 \le 1/2 - 3/5$ (Not Holds)
  $n = 6$, $c_1 \le 1/2 - 3/6$ (Not Holds and Equals ZERO)
  $n = 7$, $c_1 \le 1/2 - 3/7$ or $c_1 \le (7\text{-}6)/14$ or $c_1 \le 1/14$ (Holds for $c_1 \le 1/14$)
- The inequality holds for any $n \ge 7$ and $c_1 \le 1/14$.
- Thus by choosing the constant $c_1 = 1/14$ and $n_0 = 7$, one can verify that
  $n^2/2 - 3n = \Omega(n^2)$ holds.

Asymptotic Notation: Omega Notation (Ω)

- Consider $f(n) = 2n^2 + 5$, $g(n) = 7n$. Find some constant c such that $f(n) \geq g(n)$
- For n=0, f(n) = 0+5=5, g(n) = 0 -> f(n)>g(n)
- For n=1, f(n) = 2*1*1+5 = 7, g(n) = 7 -> f(n) = g(n)
- For n=3, f(n) = 2*3*3+5 = 23, g(n) = 21 -> f(n) ≥ g(n)

- Thus for n > 3, f(n) ≥ g(n) ->always lower bound.

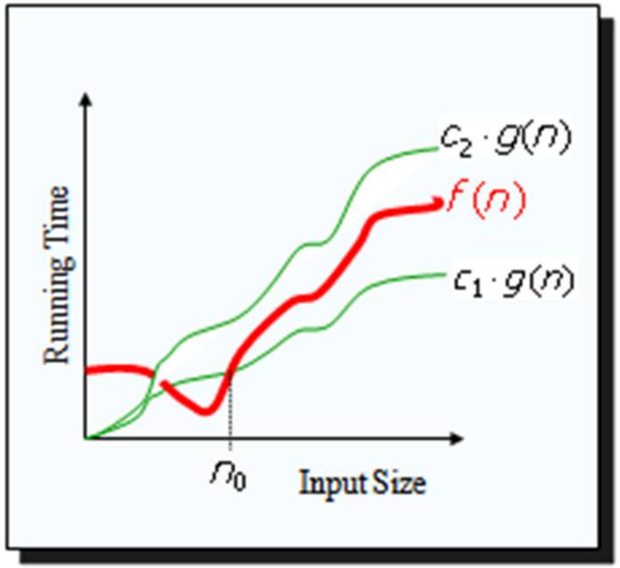
Asymptotic Notation: Theta Notation (Θ)

- Asymptotically tight bound used for average-case analysis
- Let f(n) and g(n) are functions over non-negative integers
- if there exists constants c_1, c_2, and n_0, such that
  $c_1\, g(n) \le f(n) \le c_2\, g(n)$ for $n \ge n_0$
- $f(n)=\Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- Example $f(n) = 18n+9$, $c_1\, g(n) = 18n$, $c_2\, g(n) = 27n$

  $f(n) > 18n$ and $f(n) \le 27n$

so we can say $f(n) = O(n)$ and $f(n) = \Omega(n)$

i.e $f(n) = \Theta(n)$



Example: Theta

- Show that: $n^2/2 - 3n = \theta(n^2)$
- Determine positive constants c_1, c_2, and n_0 such that

$$c_1 n^2 \le n^2/2 - 3n \le c_2 n^2 \text{ for all } n \ge n_0$$

- Diving by n^2

$$c_1 \le 1/2 - 3/n \le c_2$$

- Right Hand Side Inequality holds for any $n \ge 1$ and $c_2 \ge 1/2$.
- Left Hand Side Inequality holds for any $n \ge 7$ and $c_1 \le 1/14$.
- Thus by choosing the constants $c_1 = 1/14$ and $c_2 = 1/2$

  and $n_0 = 7$, one can verify that $n^2/2 - 3n = \theta(n^2)$ holds.

o-Notation

- o-notation denotes an upper bound that is not asymptotically tight.
- Formally o(g(n)) ("little-oh of g of n") is defined as the set

$$o(g(n)) = \{ f(n) : \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0 \} .$$

- For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.
- Intuitively, in o-notation, the function f(n) becomes insignificant relative to g(n) as n approaches infinity; that is,

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

ω-Notation

- ω-notation denotes a lower bound that is not asymptotically tight.
- One way to define it is as f(n) ∈ ω(g(n)) if and only if g(n) ∈ o(f(n)).
- Formally, ω(g(n)) (“little-omega of g of n”) is defined as the set

$$\omega(g(n)) = \{ f(n) : \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0 \} .$$

- For example, $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$.
- The relation f(n) ∈ ω(g(n)) implies that limit $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$, if exists.

Precedence of the complexity

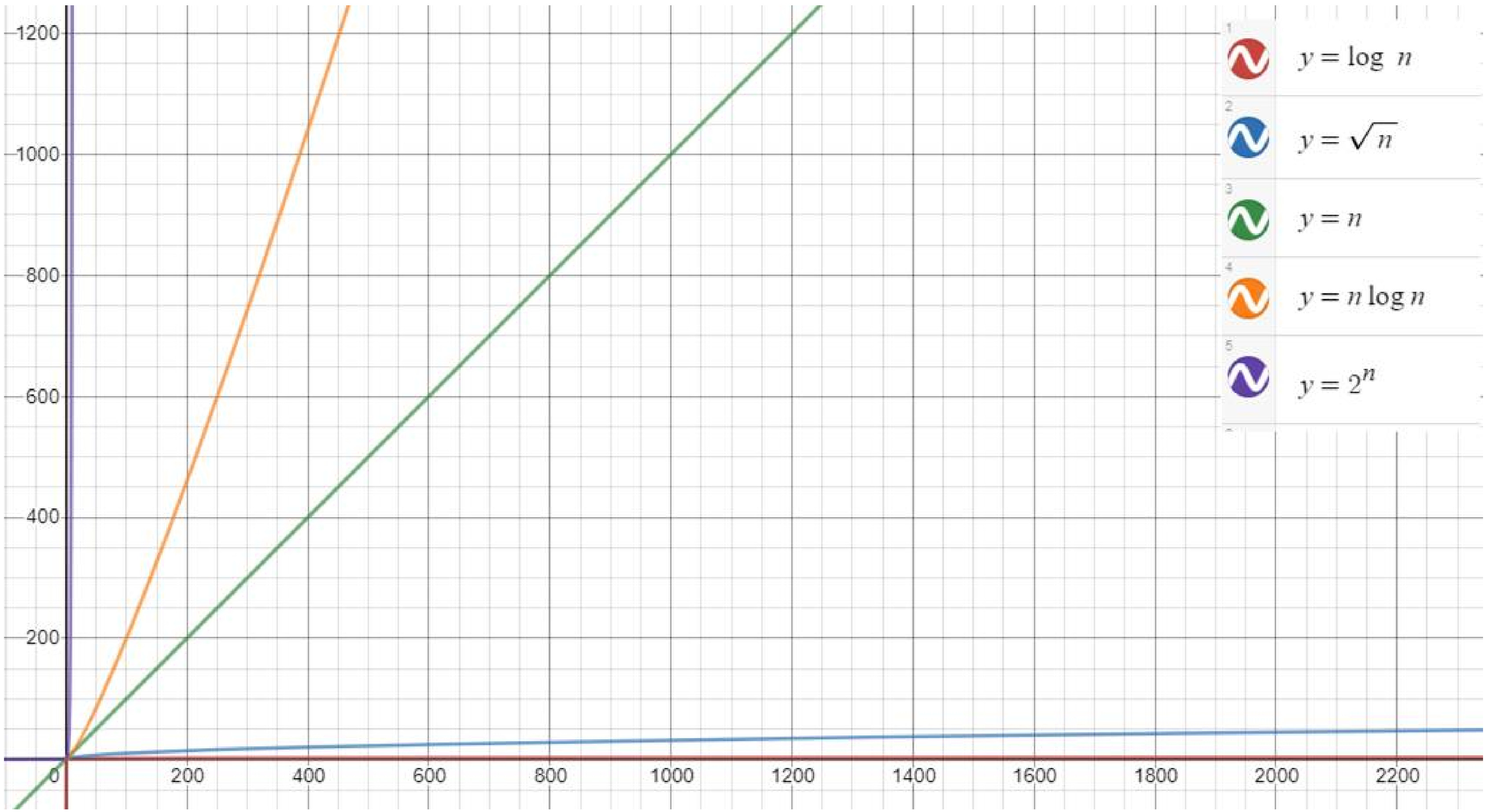
Easy

$$\boxed{c < \log^k N < N < N \log N < N^2 < N^3 <}$$

$$\boxed{2^N < 3^N < N! < N^N}$$

Hard

We can make a distinction between problems that have polynomial time algorithms and those that have algorithms that are worse than polynomial time.



1200
1000
800
600
400
200
0
200
400
600
800
1000
1200
1400
1600
1800
2000
2200
$y = \log n$
$y = \sqrt{n}$
$y = n$
$y = n \log n$
$y = 2^n$

Frequency Count

- Complexity of algorithm is calculated using frequency count of each statement.
- Frequency count defines the number of times that statement is executed in the program with respect to input.

Construct	code	FC
Statement s	i=0; a =b+c;	1

If else	If (a<b)
	{ statements; }
	Else
	{ statements; }
		Largest block statement counts

For, while, do while	for(i=1;i<=n;i++)
	{ }
	
	i=1;
	While (i<=n)
	{ }
	
	Do { }
	While(i<=n)
		n+1
		
		n+1
		
		n+1

Frequency Count

Example -1

Code	FC	Reason
for(i=1; i<=n; i++)	n+1	execute 1 to n times for true and 1 more time for false
{	-	ignore
x =x +1;	n	execute whenever you will enter into the loop that means only for true condition. So loop is true from 1 to n times
}		ignore
Total F(n) =	=n+1+n =2**n**+1	TC = O(n)

```
for(i=1;i<=5;i++)
{
   x =x+1
}
```

i = 1 -> x = x+1, i=i+1 -> i = 2

i = 2 -> x = x+1, i=i+1 -> i = 3

i = 3 -> x = x+1, i=i+1 -> i = 4

i = 4 -> x = x+1, i=i+1 -> i = 5

i = 5 -> x = x+1, i=i+1 -> i = 6

i = 6 checks condition, exits for loop

Frequency Count

Example -2

Code	FC	Reason
for(i=1; i<=n; i++)	n+1	execute 1 to n times for true and 1 more time for false
{	-	ignore
for(i=1; i<=m; i++)	n(m+1)	execute 1 to m times for true and 1 more time for false i.e m+1 for every true condition of outer loop. So outer loop gives n times true i.e n(m+1)
{		ignore
x =x +1;	nm	Inner loop is true m times per true iteration of outer loop. So outer loop is true n times so nm times this statement will execute
}}		ignore
Total F(n) =	= n+1+n(m+1)+mn = n+1+nm+n+mn = 2nm+2n+1	TC = O(mn)

Example -3

```
A(n)
{
for(i=1;i<=n,i++)        ⟶     n
   for(j=1;j<=n;j++)     ⟶     n
printf("hi");
}
```

Total Time Complexity – n*n = $O(n^2)$

Example -4

```
for(i=1; i<=n; i++)                - n
{
   for(j=1; j<=n; j++)             - n
  {
    for(k=1; k<=n; k++)            - n
    {
       a = a * b +c;
     }
  }
}
```

Total Time complexity = $O(n^3)$

Example -5

```
A(n)
{
for(i=1;i<n;i=i*2)
    printf("hi");
}
```

for the nth value	**1**	**2**	**4**	**8**	**............**	**n**
Loop will run power of 2's times	2^0	2^1	2^2	2^3	**..........**	2^k

$2^k = n$ ➡ $k = \log_2 n$ T.C. = $O(\log_2 n)$

Example -6

```
A(n)
{
while(n>1)
{
    n=n/2;
}
}
```

Assume n>=2

n =	**2**	**4**	**8**	**............**	**n**
Loop will run	**1**	**2**	**3**	**...........**	$\mathbf{2^k}$

$\mathbf{2^k = n} \Longrightarrow k = \log_2 n$ T.C. = $O(\log_2 n)$

Example -7

```
A(n)
{
int i,j,k;
for(i=n/2;i<=n,i++)              →    n/2
   for(j=1;j<=n/2;j++)           →    n/2
       for(k=1; k<=n; k=k*2)     →    log2n
     printf("hi");
}
```

Time Complexity $= n/2 * n/2 * \log_2 n$

$= O(n^2 \log_2 n)$

Example -8

`A(n)`		
`{`		
`int i,j,k;`		
`for(i=n/2;i<=n,i++)`	$\longrightarrow$	$n/2$
`    for(j=1;j<=n/2; j=2*j)`	$\longrightarrow$	$\log_2 n$
`        for(k=1; k<=n; k=k*2)`	$\longrightarrow$	$\log_2 n$
`      printf("hi");`		
`}`		

Time Complexity $= n/2 * \log_2 n * \log_2 n$

$= O(n\,(\log_2 n)^2)$

Example -9

```
A(n)
{
int i,j,k,n;
for(i=1;i<=n,i++)
    for(j=1;j<=i²; j=j++)
        for(k=1; k<=n/2; k=k++)
       printf("hi");
}
```

i=1	**i=2**	**3**	**n**
j=1 times	j=4 times	**9**	n^2
K = n/2*1 times	K = n/2*4 times	K = n/2*9 times	K = $n/2*n^2$ times

Total T.C. = $n/2*1 + n/2*4 + n/2*9 + \ldots\ldots. n/2*n^2$

$= n/2(1+4+9+\ldots\ldots n^2)$

$= n/2(n(n+1)(2n+1)/6)$

$= O(n^4)$

Example 10

$$A[i] = A[0] + A[1] + \ldots + A[i]$$

Algorithm **arrayElementSum**(A,N) Input: An array **A** containing **N** integers.

Output: An updated array **A** containing **N** integers.

1

1. **for** i = 1 to N – 1 **do**
2. sum = 0
3. **for** j = 0 to i **do**
4. sum = sum + A[j]
5. A[i] = sum

Option 2 is better

2

1. **for** i = 1 to N – 1 **do**
2. A[i] = A[i] + A[i – 1]

Contd...

	Cost	Frequency
for i = 1 to N – 1 **do**	c1	N
sum = 0	c2	N – 1
for j = 0 to i **do**	c3	$\sum_{i=1}^{N-1}(i+2)$
sum = sum + A[j]	c4	$\sum_{i=1}^{N-1}(i+1)$
A[i] = sum	c5	N – 1

$$c1N + c2(N-1) + c3\sum_{i=1}^{N-1}(i+2) + c4\sum_{i=1}^{N-1}(i+1) + c5(N-1)$$

$$c1N + c2N - c2 + c3\left(\frac{N(N-1)}{2}\right) + 2.c3.N - 2.c3 + c4\left(\frac{N(N-1)}{2}\right) + c4N - c4 + c5N - c5)$$

$$N^2\left(\frac{c3}{2} + \frac{c4}{2}\right) + N\left(c1 + c2 + \frac{3}{2}c3 + \frac{c4}{2} + c5\right) - (c2 + 2.c3 + c4 + c5)$$

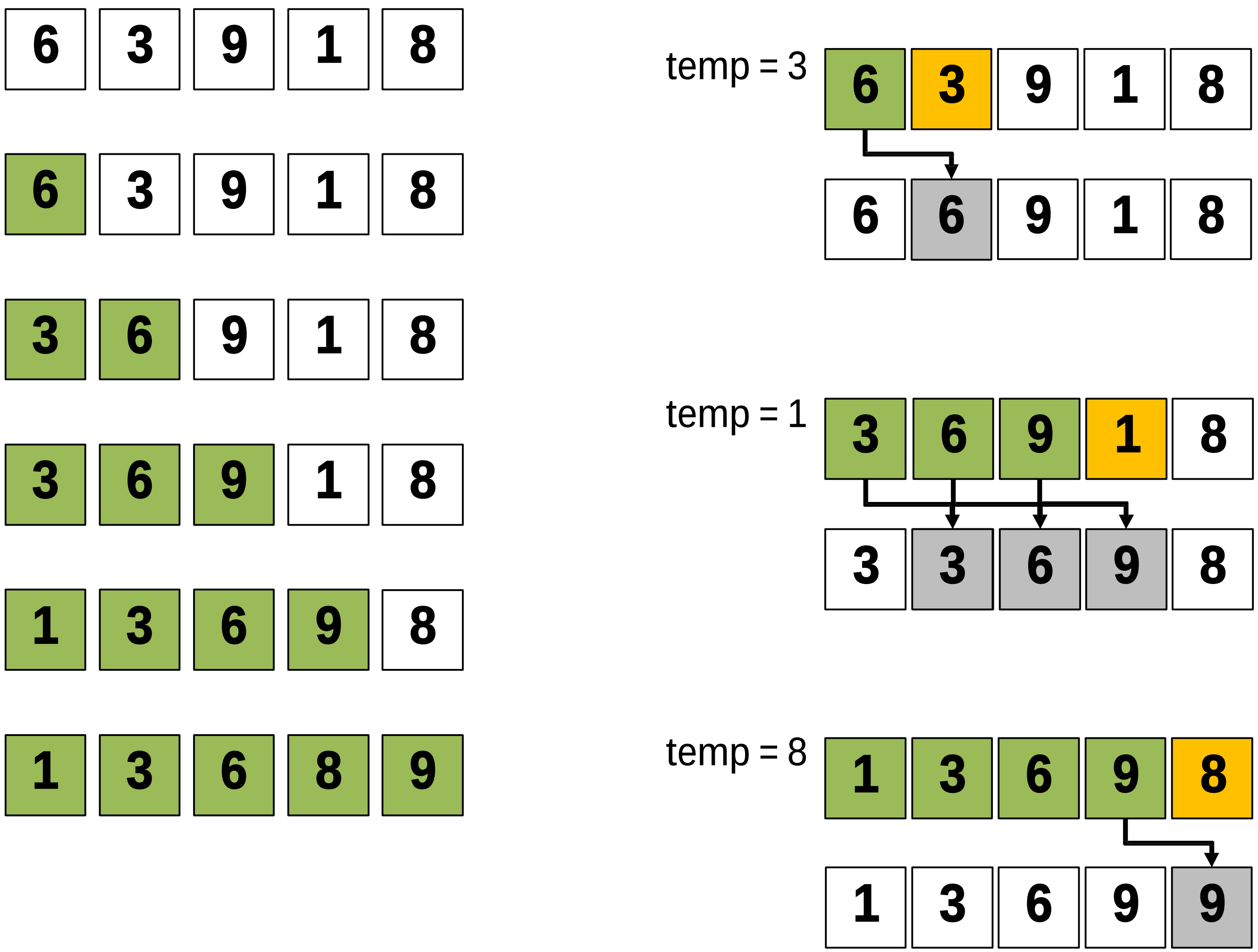
Contd…

	Cost	Frequency
1. **for** i = 1 to N – 1 **do**	c_1	N
2. A[i] = A[i] + A[i – 1]	c_2	N – 1

$$c1N + c2(N-1)$$

$$N(c1+c2) - c2$$

Example 11: Insertion Sort



Algorithm **insertionSort**(A, N)

Input: An array **A** containing **N** elements.

Output: The elements of **A** get sorted in increasing order.

1.	**for** i = 1 to N – 1	c1	N
2.	temp = A[i]	c2	N–1
3.	j = i	c3	N–1
4.	**while** j > 0 and a[j-1] > temp	c4	$\sum_{i=1}^{N-1} t_i$
5.	a[j] = a[j-1]	c5	$\sum_{i=1}^{N-1} (t_i - 1)$
6.	j = j – 1	c6	$\sum_{i=1}^{N-1} (t_i - 1)$
7.	a[j] = temp	c7	N–1

Contd...

$$T(n) = c1N + c2(N-1) + c3(N-1) + c4\sum_{i=1}^{N-1} t_i + c5\sum_{i=1}^{N-1}(t_i - 1) + c6\sum_{i=1}^{N-1}(t_i - 1) + c7(N-1)$$

```
for i = 1 to N – 1
    temp = A[i]
    j = i
    while j > 0 and a[j-1] > temp
        a[j] = a[j-1]
        j = j – 1
    a[j] = temp
```

Best case:

$$T(n) = c1N + c2(N-1) + c3(N-1) + c4(N-1) + c7(N-1)$$

$$T(n) = (c1 + c2 + c3 + c4 + c7)N - (c2 + c3 + c4 + c7)$$

Worst case:

$$T(n) = c1N + c2(N-1) + c3(N-1) + c4\left(\frac{N(N+1)}{2} - 1\right) + c5\left(\frac{N(N-1)}{2}\right) + c6\left(\frac{N(N-1)}{2}\right) + c7(N-1)$$

$$T(n) = \left(\frac{c4}{2} + \frac{c5}{2} + \frac{c6}{2}\right)N^2 + \left(c1 + c2 + c3 + \frac{c4}{2} - \frac{c5}{2} - \frac{c6}{2} + c7\right)N - (c2 + c3 + c4 + c7)$$