Design and Analysis of Algorithms

Dina El-Manakhly, Ph. D.

dina_almnakhly@science.suez.edu.eg

Loop steps with Multiplication and Division (Example 1)

1 2 4

```
for(int i=1;i<n;i*=2)
{
  cout<< i;
}
```

3

If n is a power of two number, How many times statement 1, 2, 3, 4 are executed?

A power of two is a number of the form 2^x , x is an integer, number two is the base and integer x is the exponent.
The first ten powers of 2: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512

Example 1 (cont'd)

```
for(int i=1;i<n;i*=2)
{
  cout<< i;
}
```

If n is a power of two, How many times statement 1, 2, 3, 4 are executed?

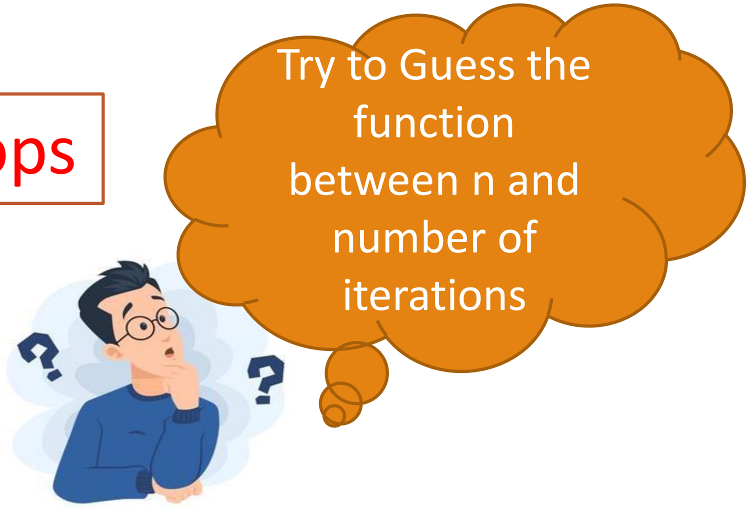
#	Times
1	
2	
3	
4	

Test: $n=32= 2^5$

i=1	i<32?	T
i=2	i<32?	T
i=4	i<32?	T
i=8	i<32?	T
i=16	i<32?	T
i=32	i<32?	F

6 iterations

n=32 ⟶ 6 loops



Example 1 (cont'd)

```
      1       2       4
for(int i=1;i<n;i*=2)
{
      3
  cout<< i;
}
```

If n is a power of two, How many times statement 1, 2, 3, 4 are executed?

#	Times
1	1
2	(log(n))+1
3	log(n)
4	log(n)

Test: $n=32= 2^5$

i=1	i<32?	**T**
i=2	i<32?	**T**
i=4	i<32?	**T**
i=8	i<32?	**T**
i=16	i<32?	**T**
i=32	i<32?	**F**

6 iterations

$32= 2^5 \longrightarrow \log_2 32=5 \longrightarrow \log 32=5$

$n=32 \longrightarrow$ 6 loops= log32+1

An exponential function has the form $n=2^x$, The inverse of this function is called the logarithm base 2, denoted $x=\log_2(n)$

Assume that each statement **(CPU)** takes time **c,** where **c** is a constant.

F(n)=T(n)= c*(1+ (log(n))+1 + log(n) + log(n)) = c*(2 + 3log(n))

Example 2

1 2 4

```
for(int i=n;i>1;i/=2)
{
  cout<< i;
}
```

3

If n is a power of two, How many times statement 1, 2, 3, 4 are executed?

#	Times
1	1
2	(log(n))+1
3	log(n)
4	log(n)

Test: $n=8=2^3$

i=8 i>1?	**T**
i=4 i>1?	**T**
i=2 i>1?	**T**
i=1 i>1?	**F**

4 iterations

$8 = 2^3 \longrightarrow \log_2 8 = 3 \longrightarrow \log 8 = 3$

$n=8 \longrightarrow$ 4 loops = log 8 +1

Assume that each statement **(CPU)** takes time **c,** where **c** is a constant.
F(n)=T(n)= c*(1+ (log(n))+1 + log(n) + log(n)) = c*(2 + 3log(n))

Example 3

```
for(int i=1;i<n;i*=3)
{
 cout<< i;
}
```

If n is a power of three, How many times statement 1, 2, 3, 4 are executed?

#	Times
1	1
2	$(\log_3(n))+1$
3	$\log_3(n)$
4	$\log_3(n)$

Test: $n=81=3^4$

i=1	i<81?	**T**
i=3	i<81?	**T**
i=9	i<81?	**T**
i=27	i<81?	**T**
i=81	i<81?	**F**

5 iterations

$81 = 3^4 \longrightarrow \log_3 81 = 4$

$n=81 \longrightarrow 5 \text{ loops} = \log_3 81 + 1$

Assume that each statement **(CPU)** takes time **c,** where **c** is a constant.

$F(n)=T(n)= c^*(1+ (\log_3(n))+1 + \log_3(n) + \log_3(n)) = c^*(2 + 3\log_3(n))$

Time complexity using summation (Example 4)

```
for(int i=0;i<n;i++)
{
 for(int k=0;k<i;k++)
 {
    cout<<k;
 }
}
```

i=0
k=0

i=1
k=0
k=1

i=2
k=0
k=1
k=2

Example 4

```
for(int i=0;i<n;i++)
{
  for(int k=0;k<i;k++)
  {
     cout<<k;
  }
}
```

		T	F
n=4	i=	0 1 2 3	4

i=0

K=0 k<i → 1 False

i=1

K=0,1 k<i → 2, 1 T+ 1 F

i=2

K=0,1,2 k<i → 3, 2 T+ 1 F

i=3

K=0,1,2,3 k<i → 4, 3 T+ 1 F

- ✓ If n=4 then number o f times (k<i) is executed= 1+2+3+4.
- ✓ In general, number o f times (k<i) is executed= 1+2+3+4+…..+n.

Example 4

```
for(int i=0;i<n;i++)
{
  for(int k=0;k<i;k++)
  {
    cout<<k;
  }
}
```

Boxed parts: 1 = `int i=0`, 2 = `i<n`, 7 = `i++`, 3 = `int k=0`, 4 = `k<i`, 6 = `k++`, 5 = `cout<<k;`

#	Times
1	1
2	n+1
3	n
4	$\frac{n(n+1)}{2}$
5	$\frac{(n-1)((n-1)+1)}{2}$
6	$\frac{(n-1)((n-1)+1)}{2}$
7	n

Statement 4: 1+2+3+4+.....+n= $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Statement 5,6: 0+1+2+3+.....+(n-1) = $\frac{(n-1)((n-1)+1)}{2}$

$$F(n)=c*(\frac{3n^2}{2} + \frac{5n}{2} + 2)$$

Time complexity using summation (cont'd)

- 1+2+3+4+……+n = $\sum_{1}^{n} i = \frac{n(n+1)}{2}$

- 2+3+4+……+n = $\sum_{2}^{n} i = (\frac{n(n+1)}{2})$-1

- 3+4+……+n = $\sum_{3}^{n} i = (\frac{n(n+1)}{2})$-3

Example 5

```
for(int i=0;i<n;i++)
{
  int k=i;
  while(k>0)
    {
      k--;
    }
}
```

#	Times
1	1
2	n+1
3	n
4	$\frac{n(n+1)}{2}$
5	$\frac{(n-1)((n-1)+1)}{2}$
6	n

$F(n) = c*(n^2+3n+2)$

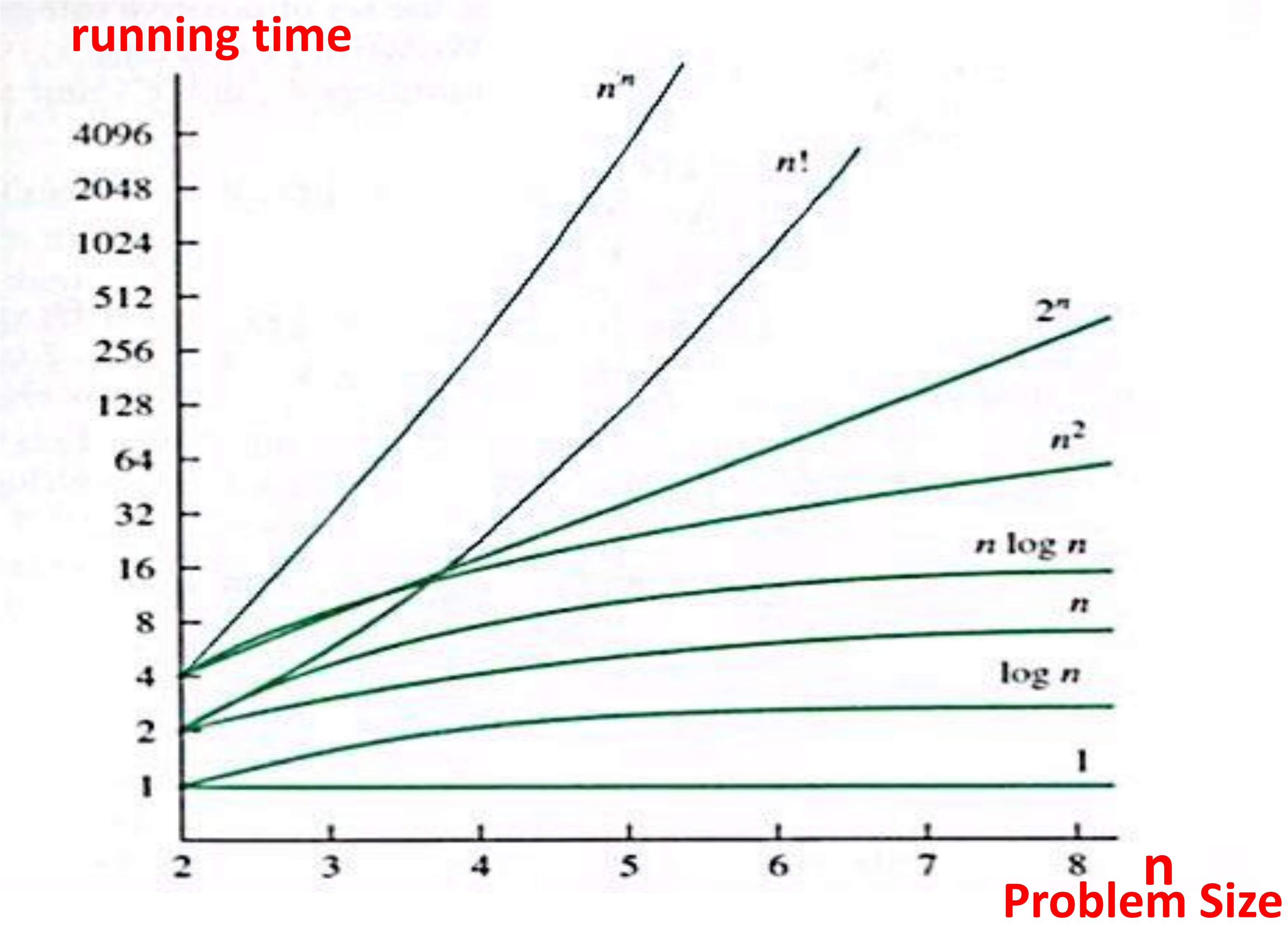
Comparison of Growth Rates

- n^n
- $n!$
- 2^n
- n^2
- $n \log n$
- n
- $\log n$
- 1

Polynomial: the base is a variable and the exponent is a constant

Exponential: the base is a constant and the exponent is a variable

Exponential growth is "bigger" and "faster" than polynomial growth



Asymptotic analysis

- Although we can sometimes determine the exact running time of an algorithm, the extra precision is not usually worth the effort of computing it.
- For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.
- When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the asymptotic efficiency of algorithms.
- Asymptotic analysis is the process of calculating the running time of an algorithm in mathematical units to find the program's limitations, or run-time performance. Big O notation is used in Computer Science to describe the performance or complexity of an algorithm.
- Big O specifically describes the worst-case scenario.

Asymptotic analysis (cont'd)

- $F(n) = n^2 + n + 5$

  Asymptotic Analysis is $O(n^2)$

- $F(n) = 5n^3 + 1000\,n^2 + 7$

  Asymptotic Analysis is $O(n^3)$

- $F(n, m) = 9n^3 + 10n^2 + 3m + 5m^2$

  Asymptotic Analysis is $O(n^3 + m^2)$

- n^n
- $n!$
- 2^n
- n^2
- $n \log n$
- n
- $\log n$
- 1

We can conclude that it is not desirable to include constants or low order terms inside a Big-Oh

Asymptotic analysis (cont'd)

- Calculate the complexity time asymptotically?

```
for (int i = 0; i <1000; i++)
    {
        cout<<i;
    }
```

This function runs in O(1) time (or "constant time").

O(1) means that it takes a constant time, like 14 nanoseconds, or three minutes, no matter the amount of input. The input size could be 1 item or 1,000 items, but this code would still just require the same time.

Asymptotic analysis (cont'd)

- Calculate the complexity time asymptotically?

```
for(int i=0;i<n;i++)
{
   cout<<i;
}
cout<<"end";
```

n+1

This function runs in O(n) time (or "linear time"). O(n) means it takes an amount of time linear with the size of the input, so an input twice the size will take twice the time.

Asymptotic analysis (cont'd)

- Calculate the complexity time asymptotically?

```
for(int i=0;i<n;i+=2)
{
   cout<<i;
}
```

$$\frac{n}{2}+1 = \frac{1}{2}n+1$$

This function runs in O(n) time (or "linear time)

Asymptotic analysis (cont'd)

- Calculate the complexity time asymptotically?

```
for(int i=0;i<n;i++)
{
   cout<<i;
}
for(int i=0;i<n;i++)
{
   cout<<i;
}
```

This function runs in O(2n), which we just call O(n)

Asymptotic analysis (cont'd)

- Calculate the complexity time asymptotically?

```
for(int i=1;i<n;i*=2)
{
 cout<< i;
}
```

$\boxed{(\log(n))+1}$

This function runs in O(log n) time

Asymptotic analysis (cont'd)

- Calculate the complexity time asymptotically?

```
for(int i=0;i<n;i++)
{
 for(int j=0;j<n;j++)
  {
   cout<<j;
  }
}
```

$\boxed{O(n)}$

$\boxed{O(n^2)}$

Here we're nesting two loops. Outer loop runs n times and inner loop runs n times for each iteration of the outer loop, giving us n^2 total prints. Thus this function runs in $O(n^2)$ time (or "quadratic time"). If the input has 10 items, we have to print 100 times. If it has 1000 items, we have to print 1000000 times.

Asymptotic analysis (cont'd)

- Calculate the complexity time asymptotically?

```
for(int i=0;i<n;i++)
{
   cout<<i;
}

for(int i=0;i<n;i++)
{
 for(int j=0;j<n;j++)
   {
    cout<<j;
   }
}
```

Runtime is $O(n + n^2)$, which we just call $O(n^2)$

Asymptotic analysis (cont'd)

- Calculate the complexity time asymptotically?

```
for(int i=1;i<n;i*=2)
{
 for(int j=0;j<n;j++)
  {
   cout<<j;
  }
}
```

$\boxed{O(\log n)}$

$\boxed{O(n \log n)}$

Runtime is $O(n \log n)$

Asymptotic analysis (cont'd)

- Calculate the complexity time asymptotically?

```
for(int i=0;i<n;i++)     O(n)
{
    for(int k=0;k<m;k++) O(mn)
    {
        cout<<k;
    }

}
```

Runtime is O(mn)

Asymptotic analysis (cont'd)

- Calculate the complexity time asymptotically?

```
for(int i=0;i<n;i++)
{
 int k=i;
 while(k>0)
  {
     k--;
  }
}
```

$O(n)$

$O(n^2)$

Runtime is $O(n^2)$