

### 2.3 Complexity of algorithms

#### 2.3.1 Constant or $O(1)$ algorithms

## $O(1)$ Algorithm

```
void lessthan(int x, int y) {  
    bool r = (x < y) ? true:false ;  
    if (r) {  
        cout << x << " is less than " << y << endl ;  
    }else {  
        cout << x << " is NOT less than " << y << endl ;  
    }  
}
```

```
void swap(int coffee, int tea) {  
    int temp = coffee ;  
    coffee = tea ;  
    tea = temp;  
}
```

Figure 2.9: Constant algorithms

#### 2.3.2 Logarithmic or $O(\log n)$ algorithms

### log n Behavior

A bartender offers 10000\$ bet. If you win, you get 10000\$ or you should pay him 10000\$

You choose a number between 1 to 1,000,000. Bartender will tell that number in 20 guesses.

After each guess, you should tell bar attender:

1. TOO HIGH
2. TOO LOW
3. YOU(bar attender) are right.

If bartender cannot get your choosen number in 20 guesses, you will win 10000\$. Other wise, you have to pay 10000\$ to the bartender.

Will you take that bet ?

How many MAXIMUM guesses are required to answer a number between 1-10 ?

$$4 = \log_2 10$$

In general  $\log_2 N$  guesses

$$\log_2 1000000 = 19.93 = 20 \text{ guesses}$$

$$2^{20} = 1048576$$

Figure 2.10: log n algorithm

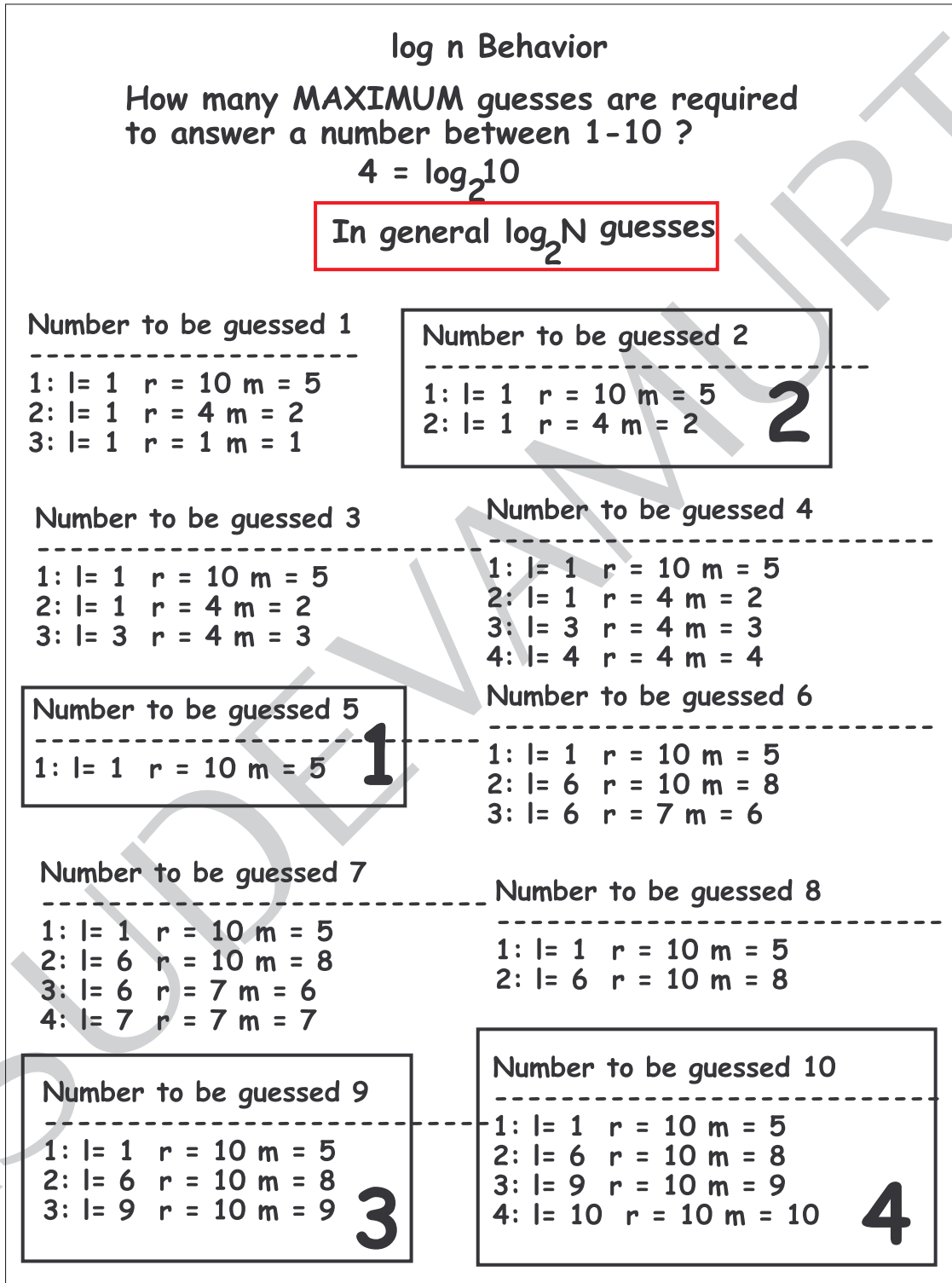


Figure 2.11: Guessing a number between 1 to 10

# O(log n) Algorithm

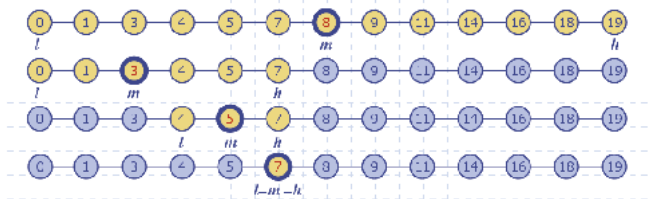
```
int iter = 0 ;
for (int i = 1; i < 1000; i = i * 2) {
    iter++ ;
    some_func() ;
}
```

```
int iter = 0 ;
for (int j = 1000; j >= 1 ; j = j / 2) {
    iter++ ;
    some_func() ;
}
```

iter	value of i	value of j
1	1	1000
2	2	500
3	4	250
4	8	125
5	16	62
6	32	31
7	64	15
8	128	7
9	256	3
10	512	1
exit	1024	0

$$f(n) = \log_2 n$$

Example: findElement(7)



## Binary search

Figure 2.12: log n algorithms

## 2.3. COMPLEXITY OF ALGORITHMS

### 2.3.3 Linear $O(n)$ algorithms

# $O(n)$ algorithms

```
int find1(int [] a, int tofind) {
    int n = a.length ;
    int j = 0 ;
    while (j < n) {
        if (a[j] == tofind) {
            return j;
        }else {
            j++ ;
        }
    }
    return -1;
}
```

**$2*n$  compare**

```
int find2(int [] a, int tofind) {
    int n = a.length ;
    assert(a[n-1] == tofind) ;
    int j = 0 ;
    while (1) {
        if (a[j] == tofind) {
            if (j == n-1) {
                return -1 ;
            }
            return j;
        }else {
            j++ ;
        }
    }
}
```

**$n+1$  compare**

Figure 2.13:  $O(n)$  algorithms

### 2.3.4 $O(n \log n)$ algorithms

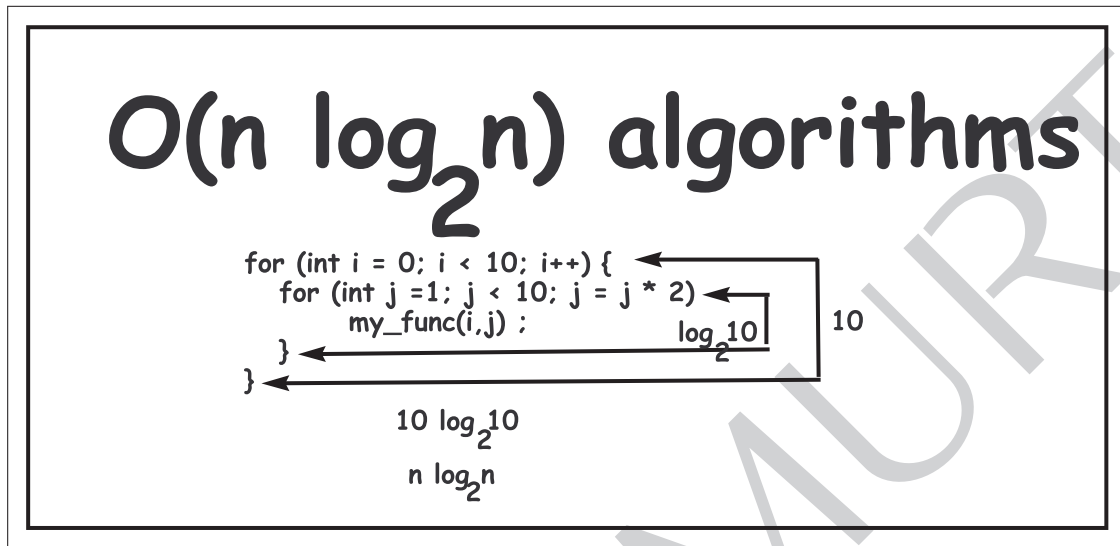


Figure 2.14:  $O(n \log n)$  algorithms

### 2.3.5 Quadratic or $O(n^2)$ algorithms

# Quadratic algorithms $O(n^2)$ algorithm)

```
for( int i = 0; i < 10; i++) {
    for (int j = 0 ; j < 10; j++) {
        func(i,j) ;
    }
}
```

outerloop executed 10 times  
each inner loop is executed 10 times

$$10 * 10 = 100$$

$$f(n) = n * n = n^2$$

```
for( int i = 0; i < 10; i++)
    for (int j = 0 ; j < i; j++) {
        func(i,j) ;
    }
}
```

outerloop executed 10 times  
each inner loop is executed i times

i	outerloop executed	innerloop executed	total
0	1	0	1
1	1	1	2
2	1	2	3
3	1	3	4
4	1	4	5
5	1	5	6
6	1	6	7
7	1	7	8
8	1	8	9
9	1	9	10

$$k = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$$

$$f(n) = \frac{n(n-1)}{2} = g(n^2)$$

Figure 2.15:  $O(n^2)$  algorithms

## Quadratic algorithms

```
void doubleLoop(int n) {  
    int k = 0 ;  
    for (int i = 0; i < n; ++i) {  
        for (int j = 0; j < n; ++j) {  
            ++k ;  
        }  
    }  
    //What is n and k  
}
```

```
void doubleLoopI(int n) {  
    int k = 0 ;  
    for (int i = 0; i < n; ++i) {  
        for (int j = 0; j < i; ++j) {  
            ++k ;  
        }  
    }  
    //What is n and k  
}
```

```
void doubleLoopC(int n, int C) {  
    int k = 0 ;  
    for (int i = 0; i < n; ++i) {  
        for (int j = 0; j < C; ++j) {  
            ++k ;  
        }  
    }  
    //What is n and k  
}
```

```
void tripleLoop(int n) {  
    int k = 0 ;  
    for (int i = 0; i < n; ++i) {  
        for (int j = 0; j < n; ++j) {  
            for (int p = 0; p < n; ++p) {  
                //What is i j and p  
                ++k ;  
            }  
        }  
    }  
    //What is n and k  
}
```

Figure 2.16: Quadratic algorithms



### 2.3.6 Exponential or $O(2^n)$ algorithms

# $O(2^n)$ algorithms

Write a program that prints truth table of n inputs

For n = 3

000      TEST your program for n = 4, 8, and 10

001

010

011      WILL this world exists for n = 64?

100      Why not?

101

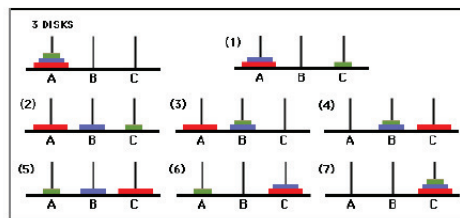
110

111

Figure 2.17: Printing a truth table

## $O(2^n)$ algorithms Tower of Hanoi

WHEN WORLD WILL COLLAPSE?



TOWER OF BRAHMA has 64 disks

To complete you require  $2^{64} - 1$  moves

18,446,744,073,709,551,615 moves

If the priests worked day and night,  
making one move every second it  
would take slightly more than 580  
billion years to accomplish the job!

Figure 2.18: Tower of Hanoi

# $O(2^n)$ algorithms Rice on chess board



This tale, be it factual or not, is often used by mathematicians to explain the concept of exponential growth. It is difficult to fathom that by using this simple formula the accumulative amount is **18,446,744,073,709,551,615** grains of rice, and that in only 64 steps.

### How much is that?

1,000,000	Million
1,000,000,000	Billion
1,000,000,000,000	Trillion
1,000,000,000,000,000	Quadrillion
1,000,000,000,000,000,000	Quintillion
1,000,000,000,000,000,000,000	Sextillion

Figure 2.19: Rice on chess board

### 2.3.7 Execution time for algorithms with given time complexities

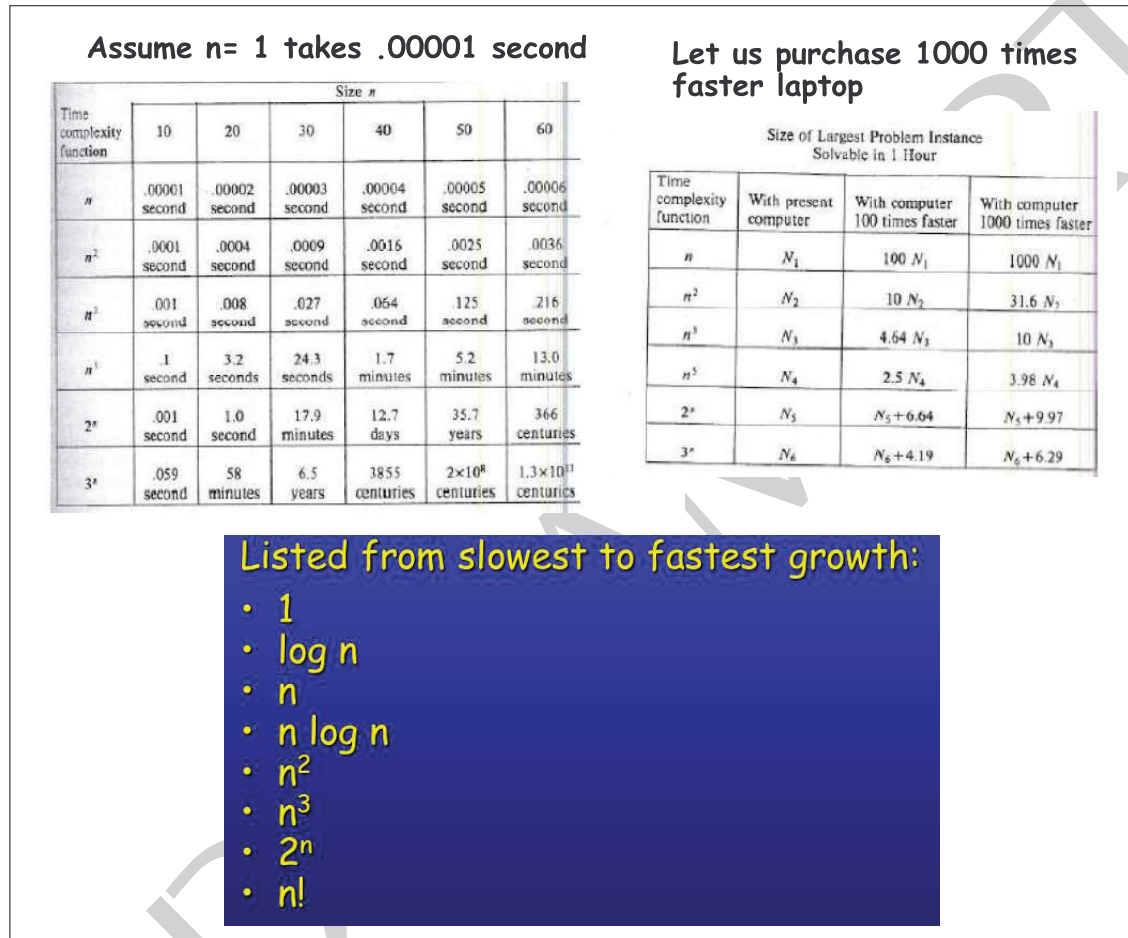


Figure 2.20: Execution time