

DATA STRUCTURES

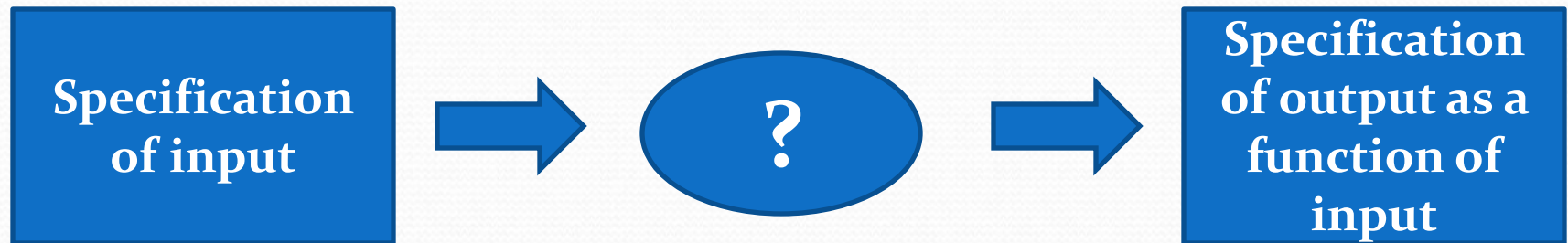
➤ Introduction:

- Basic Concepts and Notations
- Complexity analysis: time space tradeoff
- Algorithmic notations, Big O notation
- Introduction to omega, theta and little o notation

Basic Concepts and Notations

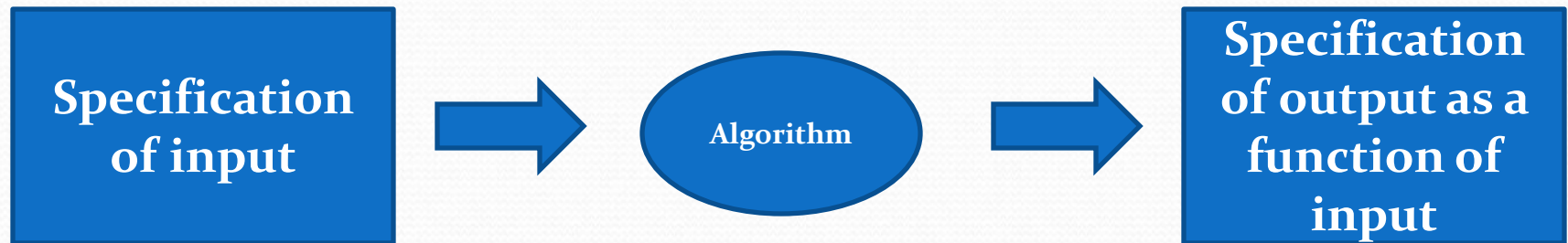
- Algorithm: Outline, the essence of a computational procedure, step-by-step instructions
- Program: an implementation of an algorithm in some programming language
- Data Structure: **Organization** of data needed to solve the problem

Algorithmic Problem



- Infinite number of input instances satisfying the specification. For example: A sorted, non-decreasing sequence of natural numbers of non-zero, finite length:
 - 1, 20, 908, 909, 100000, 10000000000.
 - 3.

Algorithmic Solution



- Algorithm describes actions on the input instance
- Infinitely many correct algorithms for the same algorithmic problem

What is a Good Algorithm?

- Efficient:
 - Running time
 - Space used
- Efficiency as a function of input size:
 - The number of bits in an input number
 - Number of data elements(numbers, points)

Complexity Analysis and Time Space Trade-off

Complexity

- A measure of the performance of an algorithm
- An algorithm's performance depends on
 - *internal* factors
 - *external* factors

External Factors

- Speed of the computer on which it is run
- Quality of the compiler
- Size of the input to the algorithm

Internal Factor

- The algorithm's efficiency, in terms of:
 - Time required to run
 - Space (memory storage) required to run

● Note:

- Complexity measures the *internal* factors (usually more interested in time than space)

Two ways of finding complexity

- Experimental study
- Theoretical Analysis

Experimental study

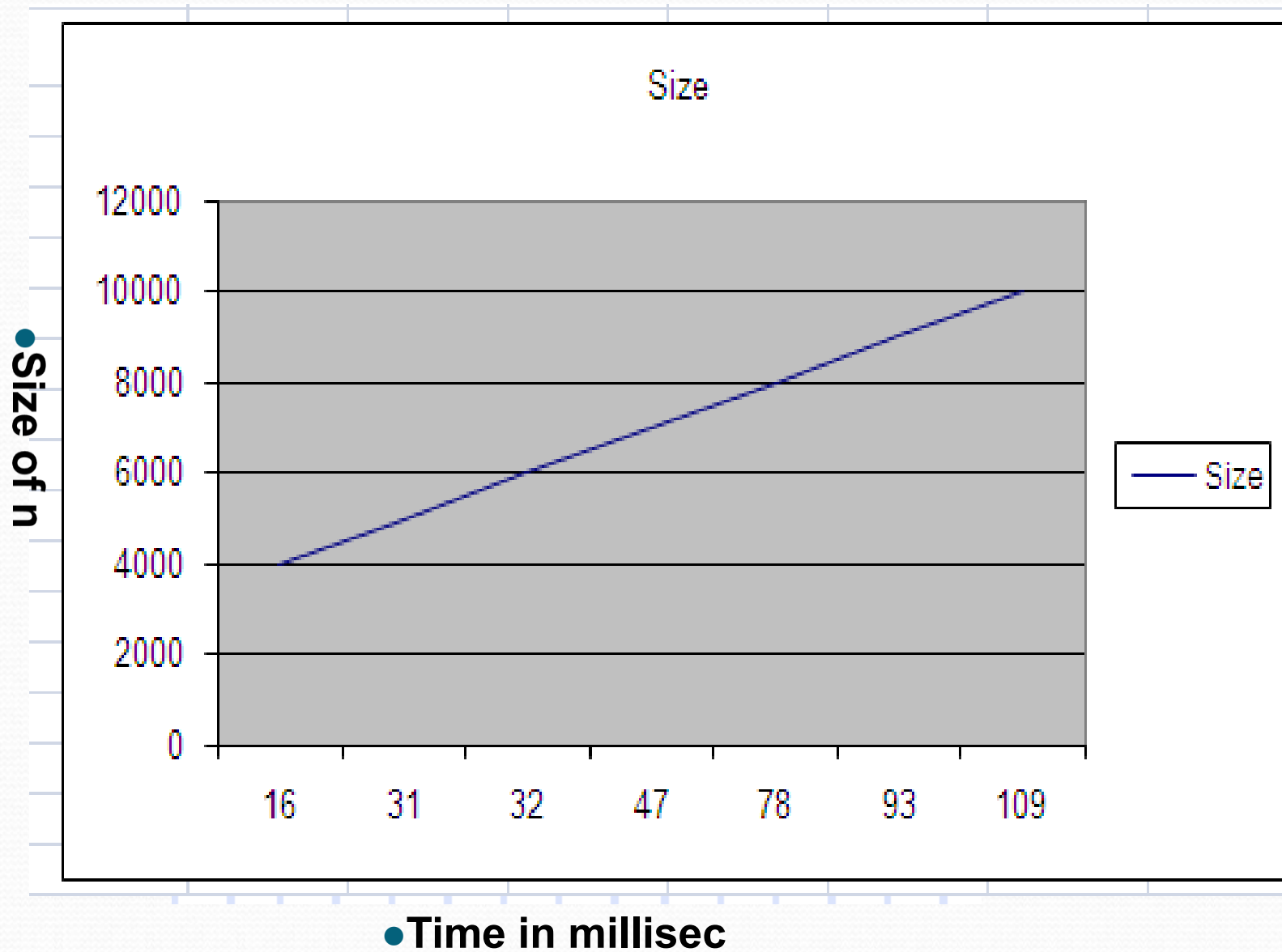
- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Get an accurate measure of the actual running time
Use a method like `System.currentTimeMillis()`
- Plot the results

Example

- a.

```
Sum=0;
for(i=0;i<N;i++)
    for(j=0;j<i;j++)
        Sum++;
```

Example graph



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Experimental data though important is not sufficient

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n .
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Complexity analysis

- Why we should analyze algorithms?
 - Predict the resources that the algorithm requires
 - Computational time (CPU consumption)
 - Memory space (RAM consumption)
 - Communication bandwidth consumption
 - The **running time** of an algorithm is:
 - The total number of primitive operations executed (machine independent steps)
 - Also known as **algorithm complexity**

Need for analysis . Internal Factors

- To determine resource consumption
 - CPU time
 - Memory space
- Compare different methods for solving the same problem before actually implementing them and running the programs.
- To find an efficient algorithm

Space Complexity

- The space needed by an algorithm is the sum of a fixed part and a variable part
- The fixed part includes space for
 - Instructions
 - Simple variables
 - Fixed size component variables
 - Space for constants
 - Etc..

Cont...

- The variable part includes space for
 - Component variables whose size is dependant on the particular problem instance being solved
 - Recursion stack space
 - Etc..

Time Complexity

- The time complexity of a problem is
 - the number of steps that it takes to solve an instance of the problem as a function of the size of the input (usually measured in bits), using the most efficient algorithm.
- The exact number of steps will depend on exactly what machine or language is being used.
- To avoid that problem, the Asymptotic notation is generally used.

Time Complexity

- Worst-case
 - An upper bound on the running time for any input of given size
- Average-case
 - Assume all inputs of a given size are equally likely
- Best-case
 - The lower bound on the running time

Time Complexity – Example

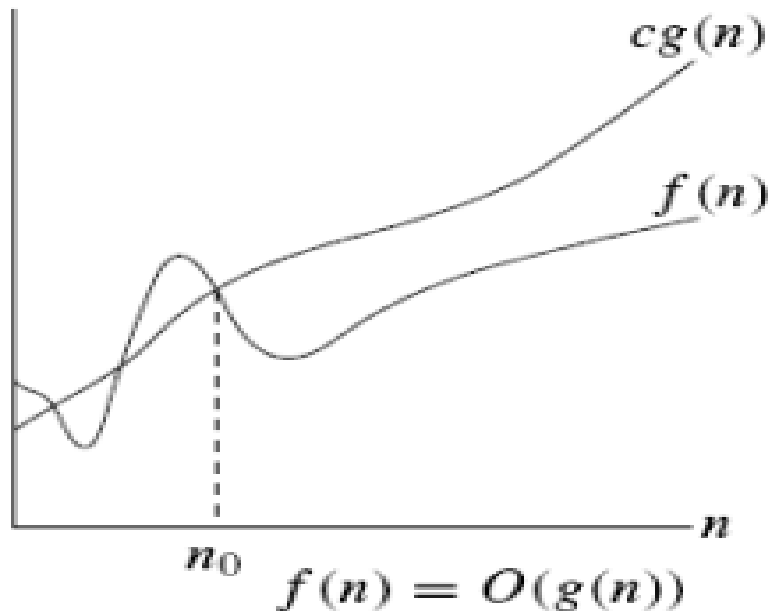
- Sequential search in a list of size n
 - Worst-case:
 - n comparisons
 - Best-case:
 - 1 comparison
 - Average-case:
 - $n/2$ comparisons

Asymptotic notations

- **Algorithm complexity** is rough estimation of the number of steps performed by given computation depending on the size of the input data
 - Measured through **asymptotic notation**
 - $O(g)$ where g is a function of the input data size
 - Examples:
 - Linear complexity $O(n)$ – all elements are processed once (or constant number of times)
 - Quadratic complexity $O(n^2)$ – each of the elements is processed n times

O-notation

Asymptotic upper bound



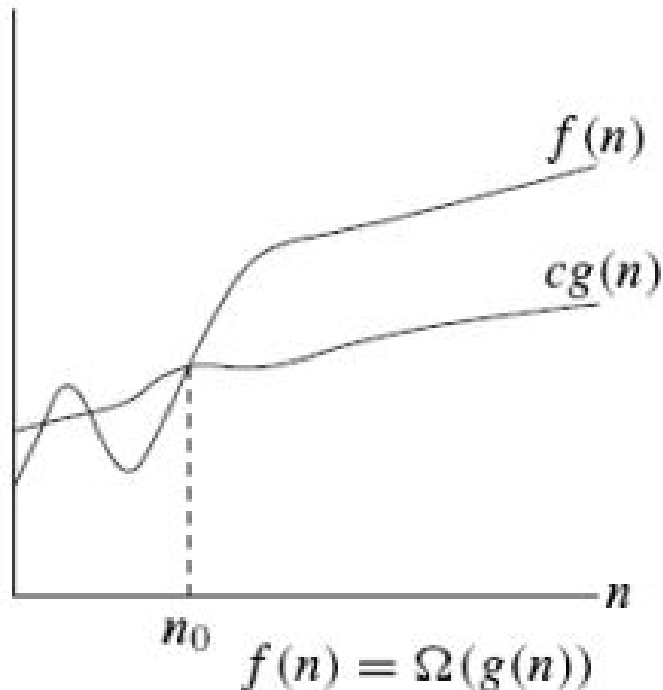
$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\} .$

Example

- The running time is $O(n^2)$ means there is a function $f(n)$ that is $O(n^2)$ such that for any value of n , no matter what particular input of size n is chosen, the running time of that input is bounded from above by the value $f(n)$.
 - $3 * n^2 + n/2 + 12 \in O(n^2)$
 - $4 * n * \log_2(3 * n + 1) + 2 * n - 1 \in O(n * \log n)$

Ω notation

Asymptotic lower bound



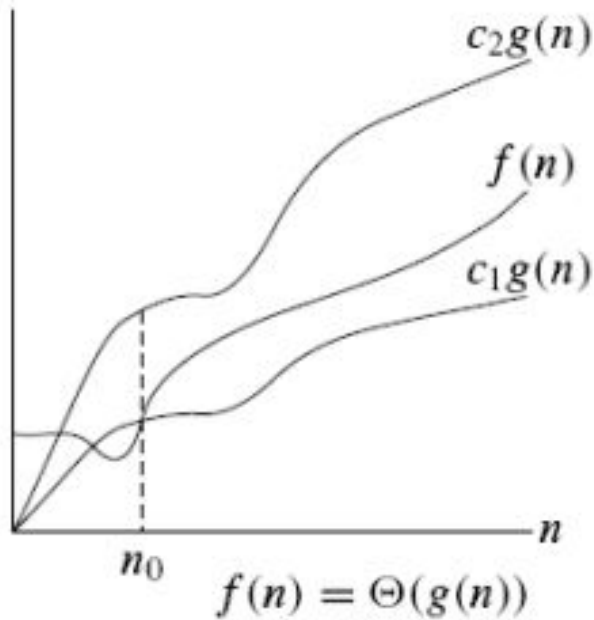
$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}.$

Example

- When we say that the running time (no modifier) of an algorithm is $\Omega(g(n))$.
- we mean that no matter what particular input of size n is chosen for each value of n , the running time on that input is at least a constant times $g(n)$, for sufficiently large n .
- $n^3 + 20n \in \Omega(n^2)$

Θ notation

$g(n)$ is an asymptotically tight bound of $f(n)$



$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$
 $0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\} .^1$

Example

$$\frac{1}{2}n^2 - 3n = \Theta(n^2)$$

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

for all $n \geq n_0$. Dividing by n^2 yields

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 .$$

$$n \geq 1, \quad c_2 \geq 1/2$$

$$n \geq 7, \quad c_1 \leq 1/14$$

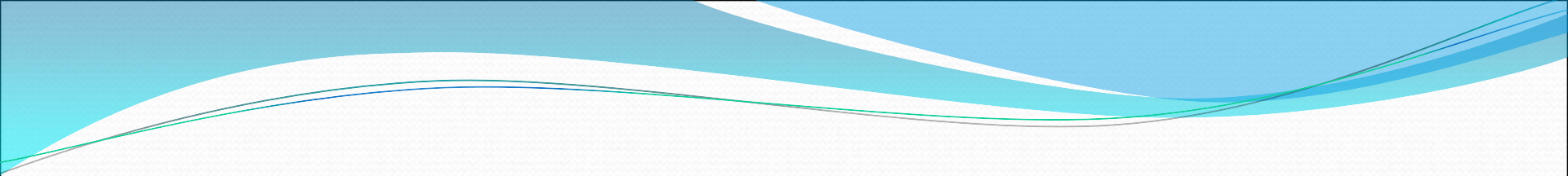
choose $c_1 = 1/14, c_2 = 1/2, n_0 = 7$.

Big O notation

- $f(n)=O(g(n))$ iff there exist a positive constant c and non-negative integer n_0 such that
$$f(n) \leq cg(n) \text{ for all } n \geq n_0.$$
- $g(n)$ is said to be an upper bound of $f(n)$.

Basic rules

1. Nested loops are multiplied together.
2. Sequential loops are added.
3. Only the largest term is kept, all others are dropped.
4. Constants are dropped.
5. Conditional checks are constant (i.e. 1).



Example of complexity

Linear loop

1)

```
for(int i = 0; i < 10; i++)  
{  
    cout << i << endl;  
}
```

//time taken = ?

2)

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

//time taken = ?

- Ans: $O(n)$

Quadratic Loops

```
1) for(int i = 0; i < 100; i++)  
    {  
        for(int j = 0; j < 100; j++)  
        {  
            //do swap stuff, constant time  
        }  
    }    //Time Taken =?
```

```
2) for(int i = 0; i < n; i++)  
    {  
        for(int j = 0; j < n; j++)  
        {  
            //do swap stuff, constant time  
        }  
    }  
    //Time Taken =?
```

- Ans $O(n^2)$

Complex condition

```
1) for(int i = 0; i < 2*100; i++)  
    {  
        cout << i << endl;  
    }
```

//Time Taken =?

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

//Time Taken =?

- At first you might say that the upper bound is $O(2n)$; however, we drop constants so it becomes $O(n)$

More loops in one program

```
1) for(int i = 0; i < 10 ; i++)  
    {  
        cout << i << endl;  
    }
```

```
for(int i = 0; i < 100; i++)  
    {  
        for(int j = 0; j < 100; j++)  
            {  
                //do constant time stuff  
            }  
    } //Time Taken =?
```

2)

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

```
for(int i = 0; i < n; i++)  
{  
    for(int j = 0; j < n; j++)  
    {  
        //do constant time stuff  
    }  
} //Time Taken =?
```

- Ans : In this case we add each loop's Big O, in this case $n+n^2$. $O(n^2+n)$ is not an acceptable answer since we must drop the lowest term. The upper bound is $O(n^2)$. Why? Because it has the largest growth rate

Quadratic loop

```
1) for(int i = 0; i < 100; i++)  
    {  
        for(int j = 0; j < 2; j++)  
        {  
            //do stuff  
        }  
    } //Time Taken =?
```

```
2) for(int i = 0; i < n; i++)  
    {  
        for(int j = 0; j < 2; j++)  
        {  
            //do stuff  
        }  
    }
```

//Time Taken =?

- Ans: Outer loop is 'n', inner loop is 2, this we have $2n$, dropped constant gives up $O(n)$

Complex iteration

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

//Time Taken =?

```
2) for(int i = 1; i < 100; i = i * 2)
{
    cout << i << endl;
}
```

//Time Taken =?

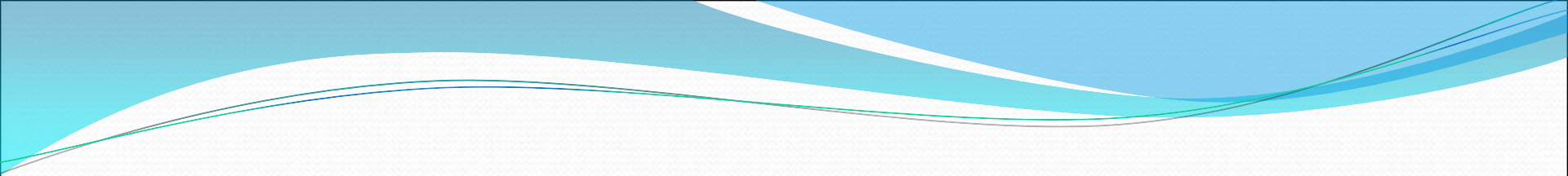
- There are n iterations, however, instead of simply incrementing, 'i' is increased by $2 \times \text{itself}$ each run. Thus the loop is $\log(n)$.

Quadratic loop

```
1) for(int i = 0; i < n; i++)  
    {  
        for(int j = 1; j < n; j *= 2)  
            {  
                //do constant time stuff  
            }  
    }
```

//Time Taken =?

- Ans: $n \cdot \log(n)$



```
While (n>=1)
```

```
{
```

```
    n=n/2;
```

```
}
```

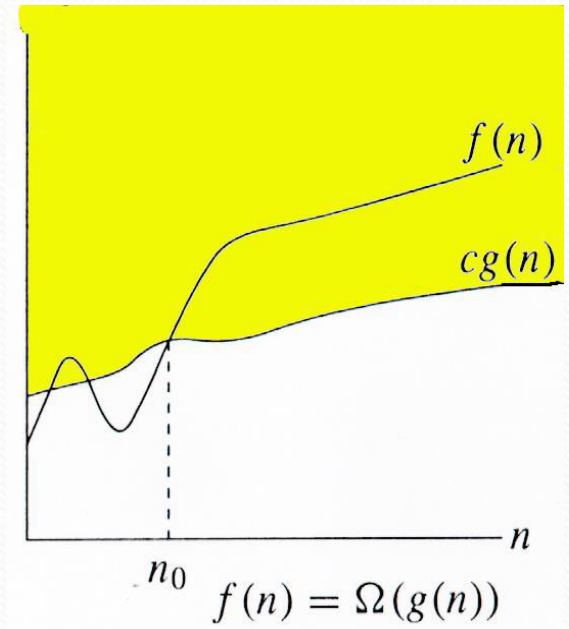
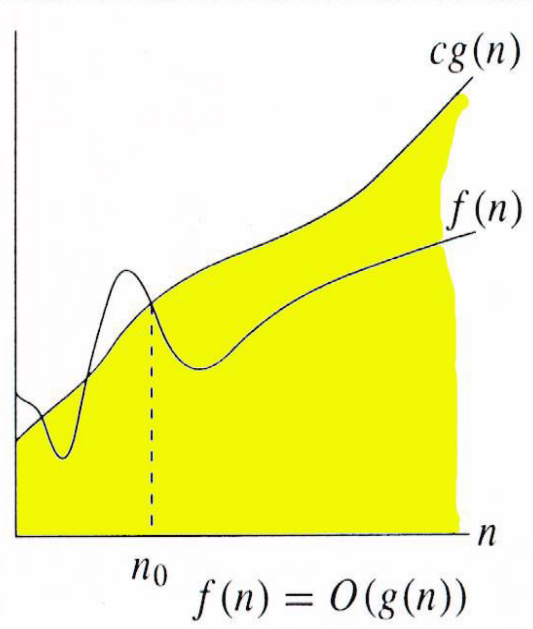
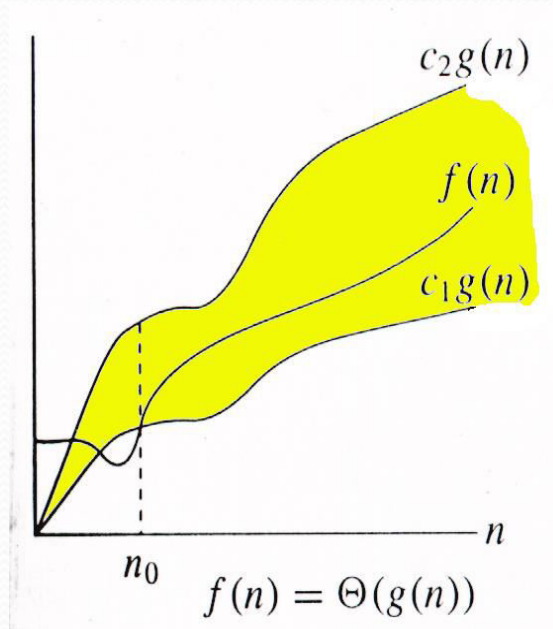
```
2) While (n>=1)
```

```
{
```

```
    n=n/2;
```

```
}
```


Relations Between Θ , O , Ω



time space tradeoff

- A time space tradeoff is a situation where the memory use can be reduced at the cost of slower program execution (and, conversely, the computation time can be reduced at the cost of increased memory use).
- As the relative costs of CPU cycles, RAM space, and hard drive space change—hard drive space has for some time been getting cheaper at a much faster rate than other components of computers[citation needed]—the appropriate choices for time space tradeoff have changed radically.
- Often, by exploiting a time space tradeoff, a program can be made to run much faster.

Time Space Trade-off

- In computer science, a **space-time** or **time-memory trade off** is a situation where the memory use can be reduced at the cost of slower program execution (or, vice versa, the computation time can be reduced at the cost of increased memory use). As the relative costs of CPU cycles, RAM space, and hard drive space change — hard drive space has for some time been getting cheaper at a much faster rate than other components of computers — the appropriate choices for space-time tradeoffs have changed radically. Often, by exploiting a space-time tradeoff, a program can be made to run much faster.

Types of Time Space Trade-off

- **Lookup tables v. recalculation**

The most common situation is an algorithm involving a lookup table: an implementation can include the entire table, which reduces computing time, but increases the amount of memory needed, or it can compute table entries as needed, increasing computing time, but reducing memory requirements.

- **Compressed v. uncompressed data**

A space-time trade off can be applied to the problem of data storage. If data is stored uncompressed, it takes more space but less time than if the data were stored compressed (since compressing the data reduces the amount of space it takes, but it takes time to run the decompression algorithm). Depending on the particular instance of the problem, either way is practical.



Thank You