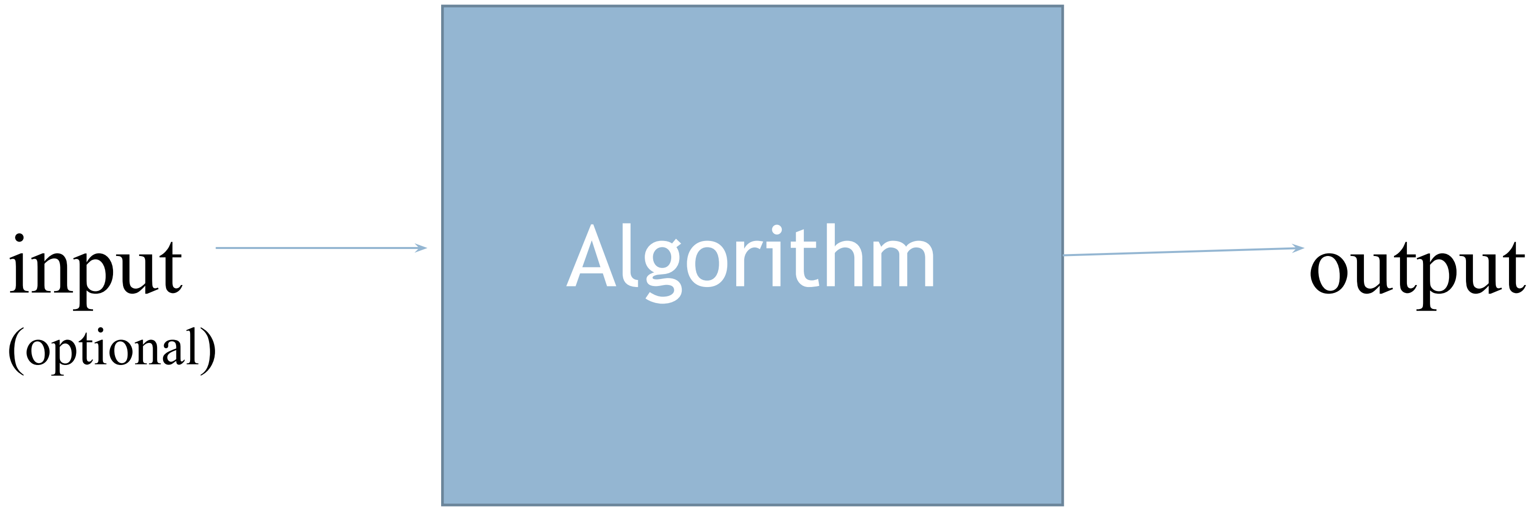
Lecture 1

(Introduction, Runtime, Asymptotic analysis, finding asymptotic notations)

What is an Algorithm?

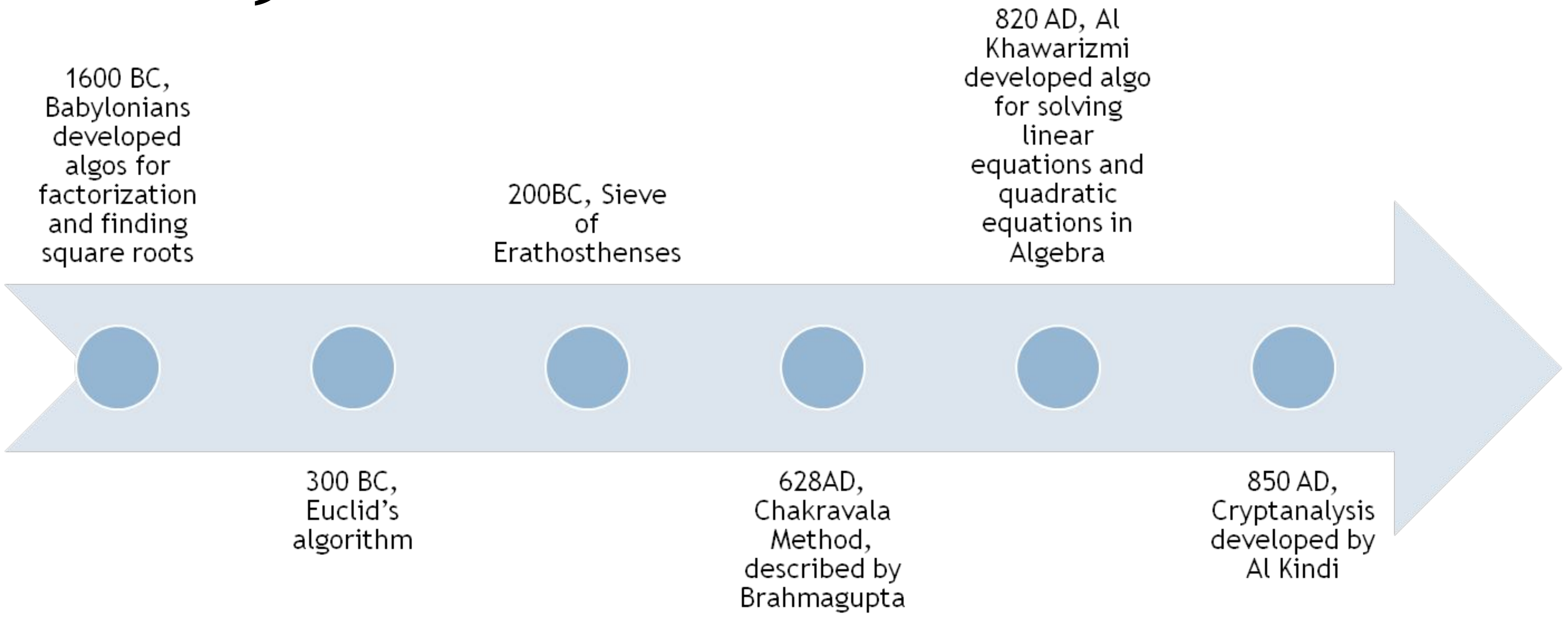
- An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.
  - It must produce correct result
  - It must finish in some finite time
  - You can represent an algorithm using pseudocode, flowchart, or even actual code

Algorithm

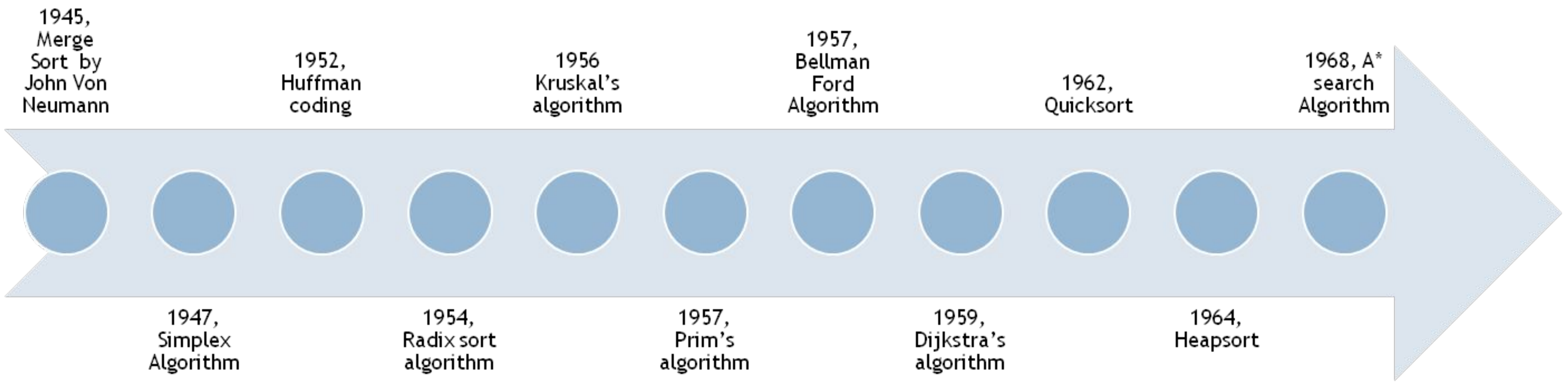


Computational procedure for solving a problem

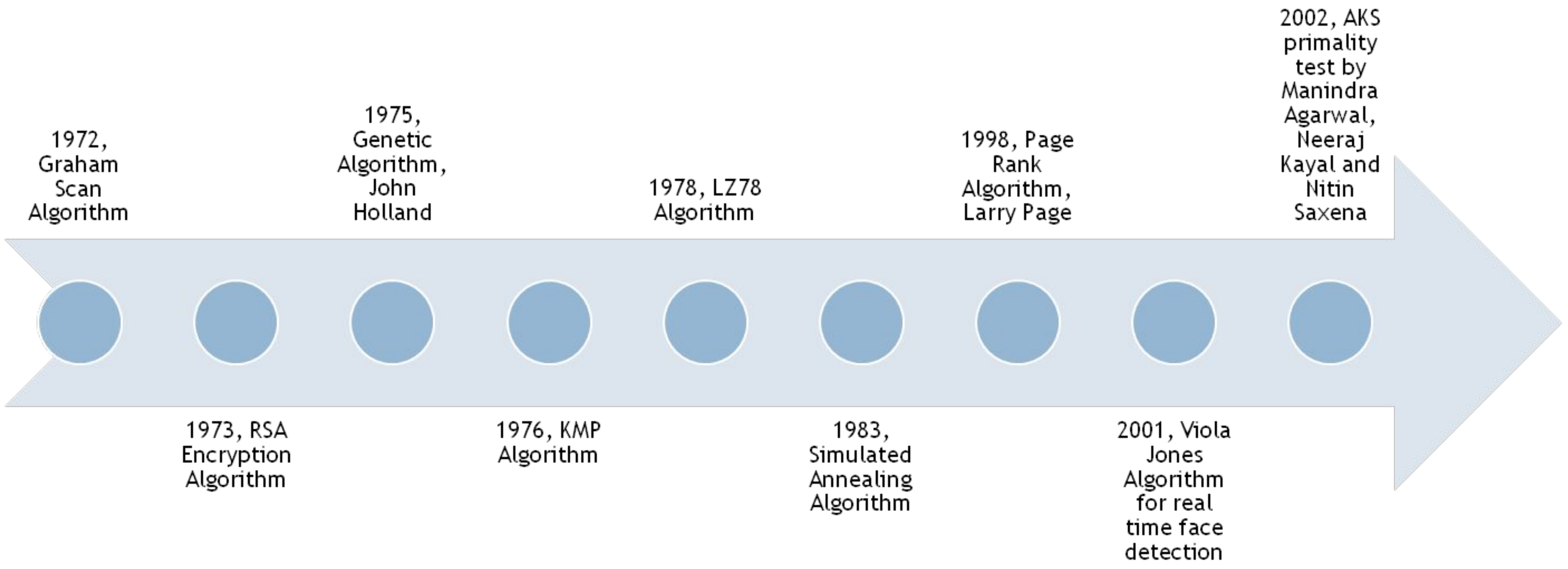
Algorithm (Brief timeline: old)


1600 BC, Babylonians developed algos for factorization and finding square roots
300 BC, Euclid's algorithm
200BC, Sieve of Erathosthenses
628AD, Chakravala Method, described by Brahmagupta
820 AD, Al Khawarizmi developed algo for solving linear equations and quadratic equations in Algebra
850 AD, Cryptanalysis developed by Al Kindi

Algorithm(Brief timeline: New Era)


1945, Merge Sort by John Von Neumann
1947, Simplex Algorithm
1952, Huffman coding
1954, Radix sort algorithm
1956 Kruskal's algorithm
1957, Prim's algorithm
1957, Bellman Ford Algorithm
1959, Dijkstra's algorithm
1962, Quicksort
1964, Heapsort
1968, A* search Algorithm

Algorithm(Brief timeline: New Era)


1972, Graham Scan Algorithm
1973, RSA Encryption Algorithm
1975, Genetic Algorithm, John Holland
1976, KMP Algorithm
1978, LZ78 Algorithm
1983, Simulated Annealing Algorithm
1998, Page Rank Algorithm, Larry Page
2001, Viola Jones Algorithm for real time face detection
2002, AKS primality test by Manindra Agarwal, Neeraj Kayal and Nitin Saxena

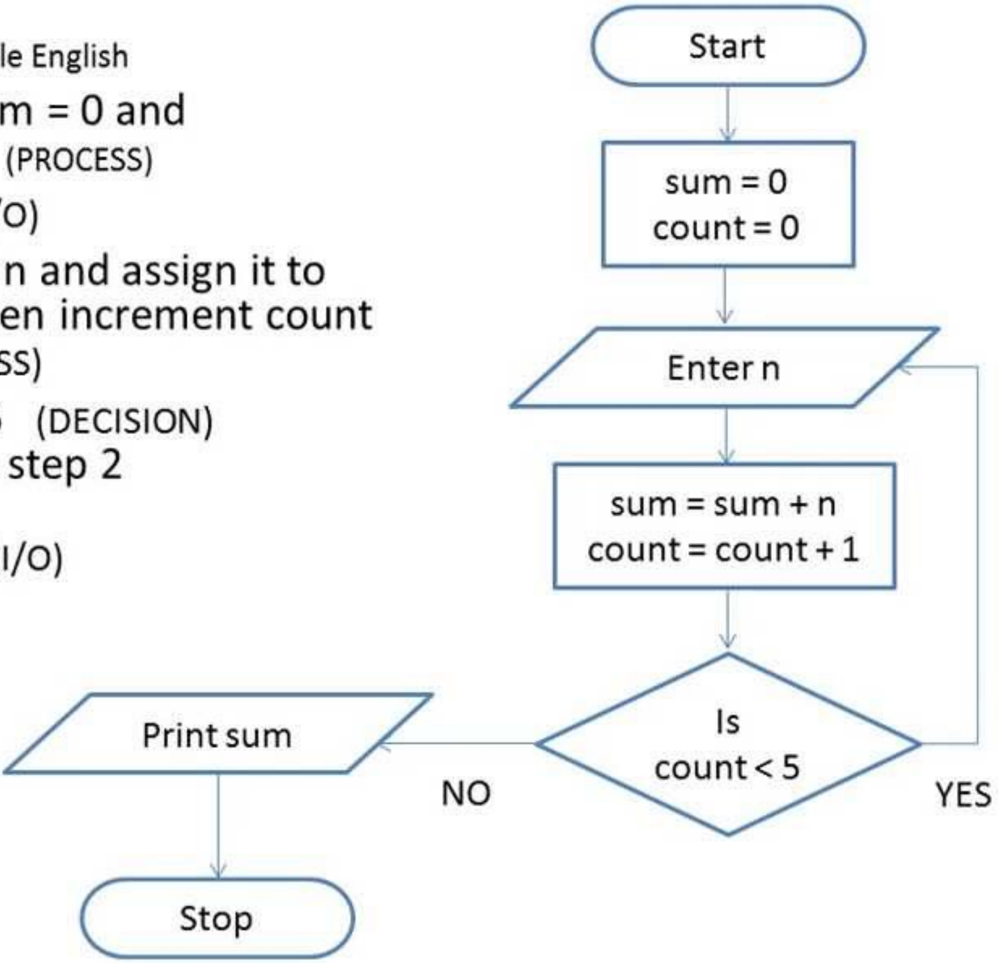
Algorithm Description

Find the sum of 5 numbers

Algorithm in simple English

1. Initialize sum = 0 and count = 0 (PROCESS)
2. Enter n (I/O)
3. Find sum + n and assign it to sum and then increment count by 1 (PROCESS)
4. Is count < 5 (DECISION)
   if YES go to step 2
   else
   Print sum (I/O)

Flowchart



Algorithm challenge 1

Can you come up with an algorithm to compute the GCD (Greatest Common Divisor) of two integers.

Algorithm Challenge 2

Can you come up with an algorithm to take an input (very long) and compute whether the input is divisible by 11 or not?

What is this course about?

The theoretical study of design and analysis of computer algorithms

Basic goals for an algorithm:

- always correct
- always terminates
- performance
   - Performance often draws the line between what is possible and what is impossible.

Design and Analysis of Algorithms

- ***Analysis:*** predict the cost of an algorithm in terms of resources and performance

- ***Design:*** design algorithms which minimize the cost

Why designing and analysis of algorithm is important?

Example:

Imagine two friends, **A**lice and **B**ob are given the task of writing an algorithm that can sort 10 million numbers

Alice writes an algorithm that takes $2N^2$ instructions and implements using computer that executes 10 billion instructions per second.

Bob writes an algorithm that takes 50NlgN instructions and implements using computer that executes only 10 million instructions per second.

Which one runs faster?

Why designing and analysis of algorithm is important?

Time required to run Alice's implementation

$$= \frac{2.(10^7)^2}{10^{10}} =$$

$20{,}000s$ (*more than* 5.5 *hours*)

Time required to run Bob's implementation

$$= \frac{50.10^7 \lg 10^7}{10^7}$$

$\approx 1163\ s$ (*less than* 20 *minutes*)

The Problem of Sorting

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 $a'_1 \le a'_2 \le \cdots \le a'_n$.

Example:

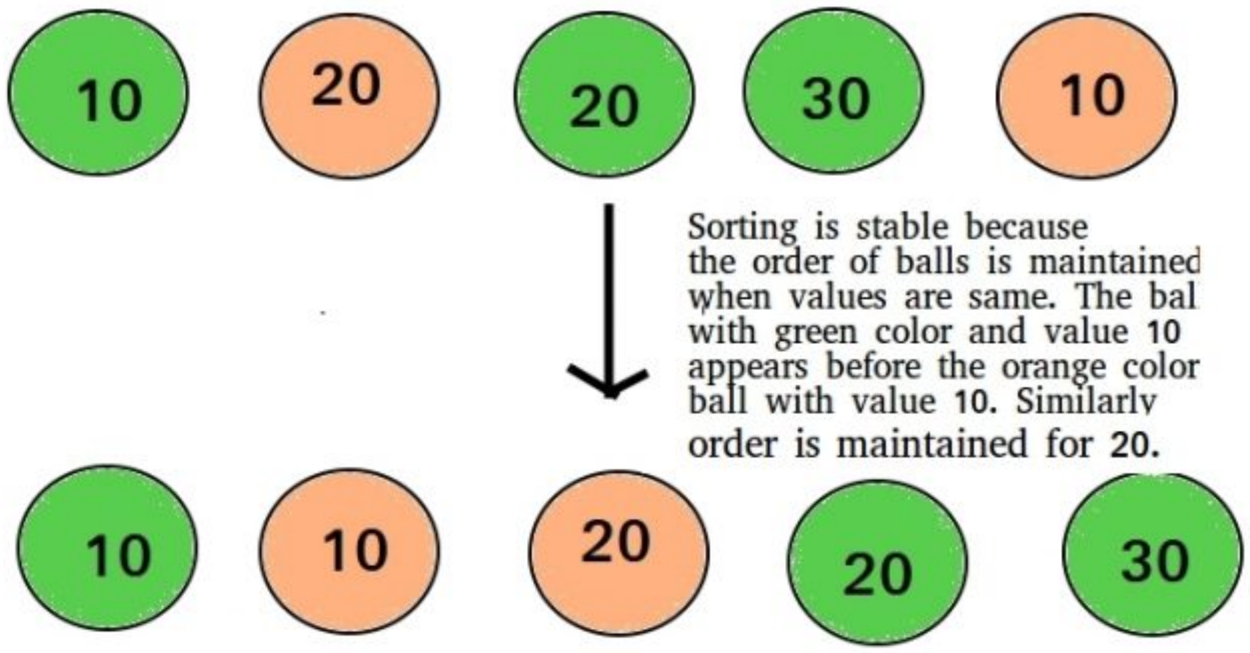
Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Sorting Algorithms

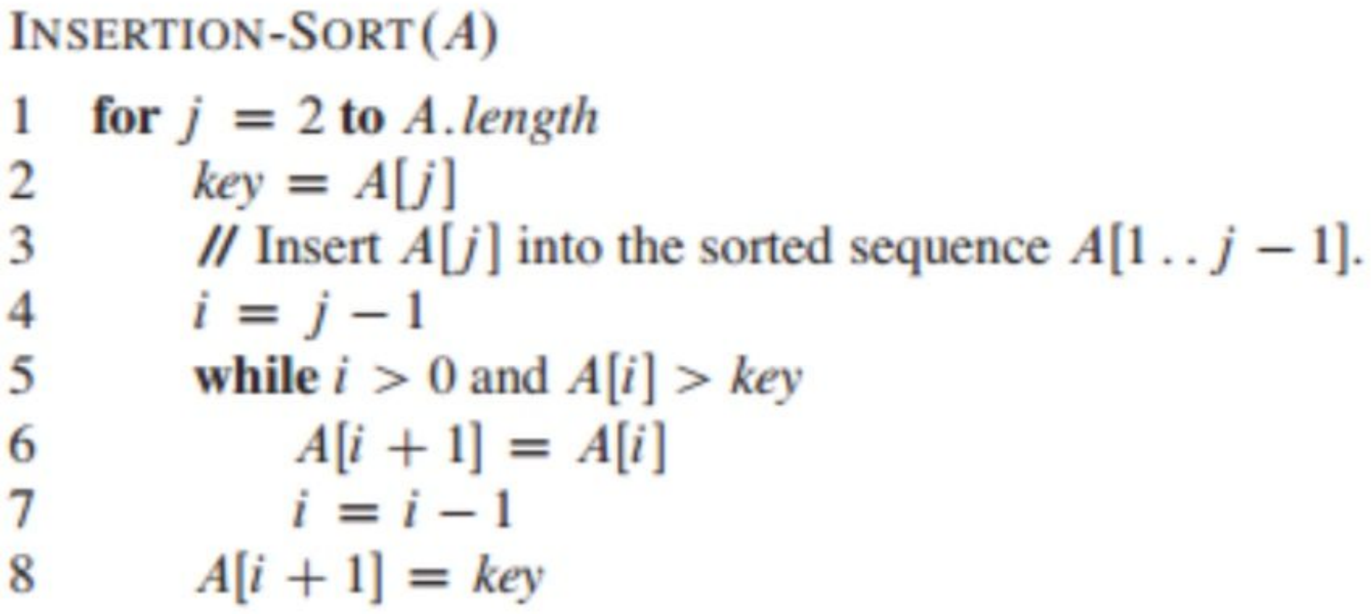
- There are various sorting algorithms including **bubble sort**, **insertion sort**, quick sort, **merge sort**, **bucket sort**, shell sort etc…
- Can either be a **stable** or an unstable sorting algorithm.
  - A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

Stability of a sort



Sorting is stable because the order of balls is maintained when values are same. The bal with green color and value 10 appears before the orange color ball with value 10. Similarly order is maintained for 20.

Insertion sort



```
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 $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

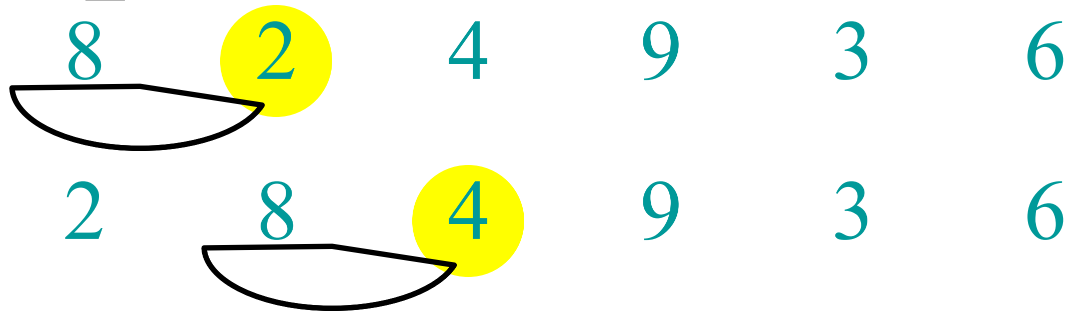
8 2 4 9 3 6

Example of insertion sort

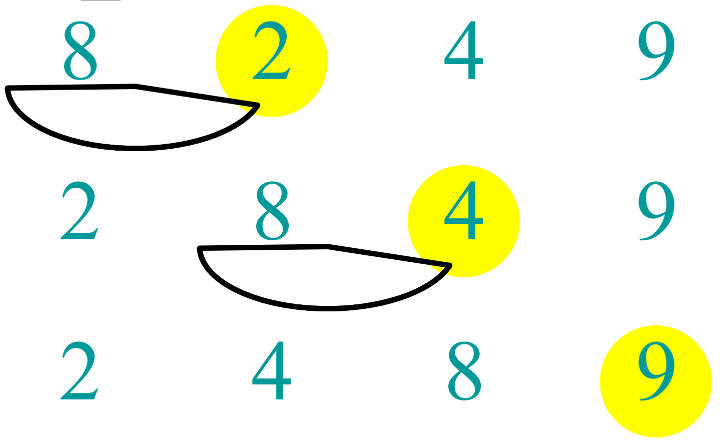
8 2 4 9 3 6

2 8 4 9 3 6

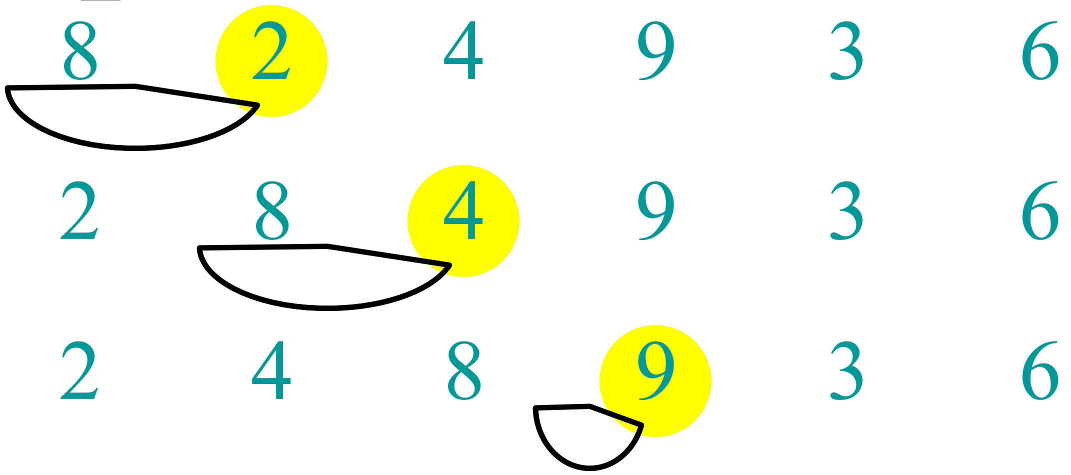
Example of insertion sort



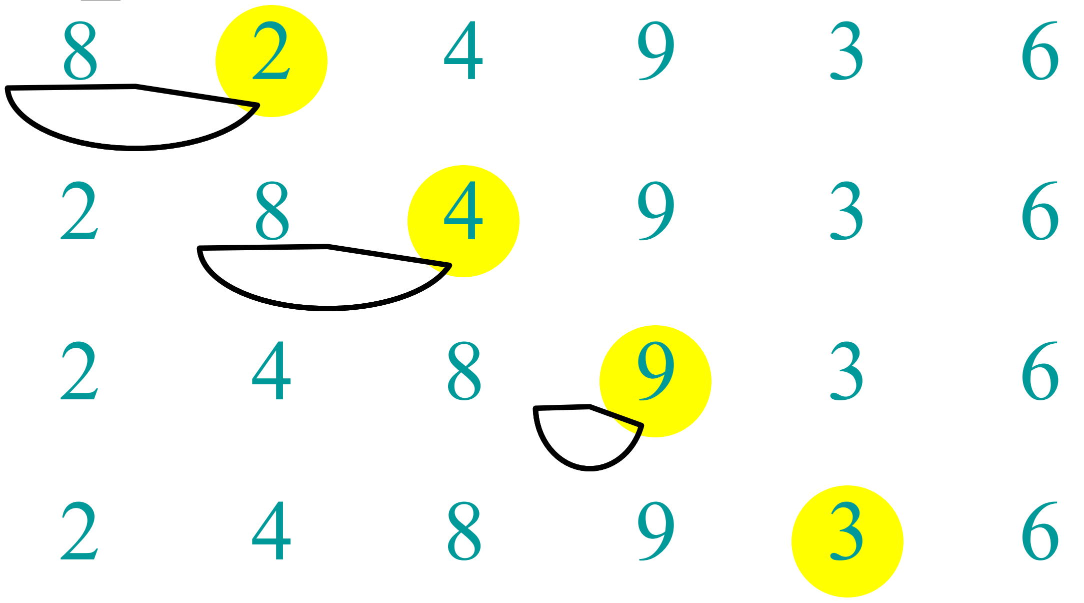
Example of insertion sort



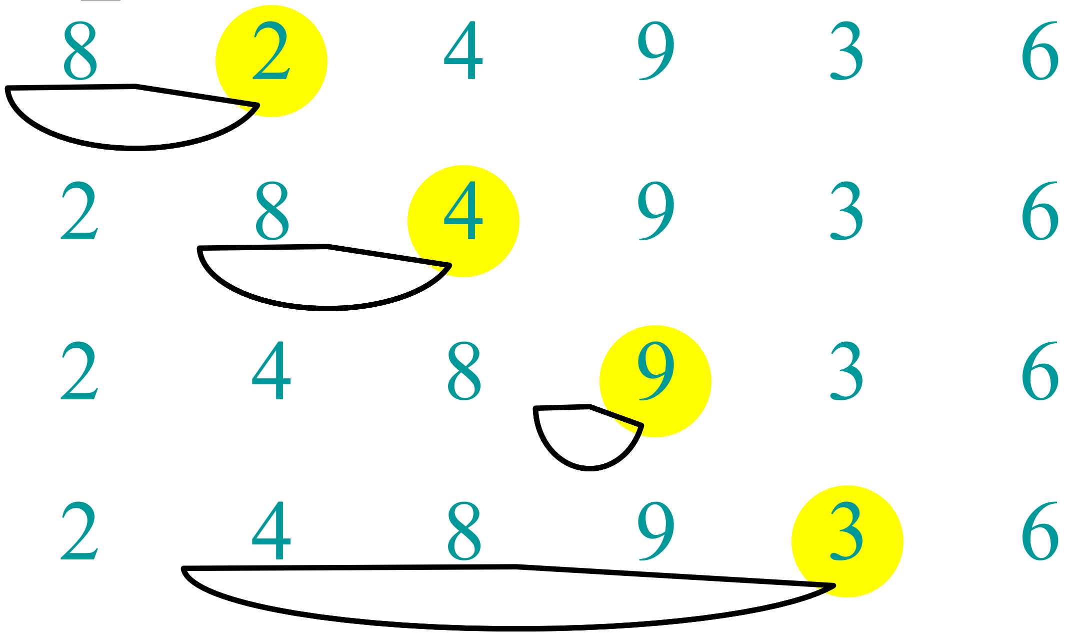
Example of insertion sort



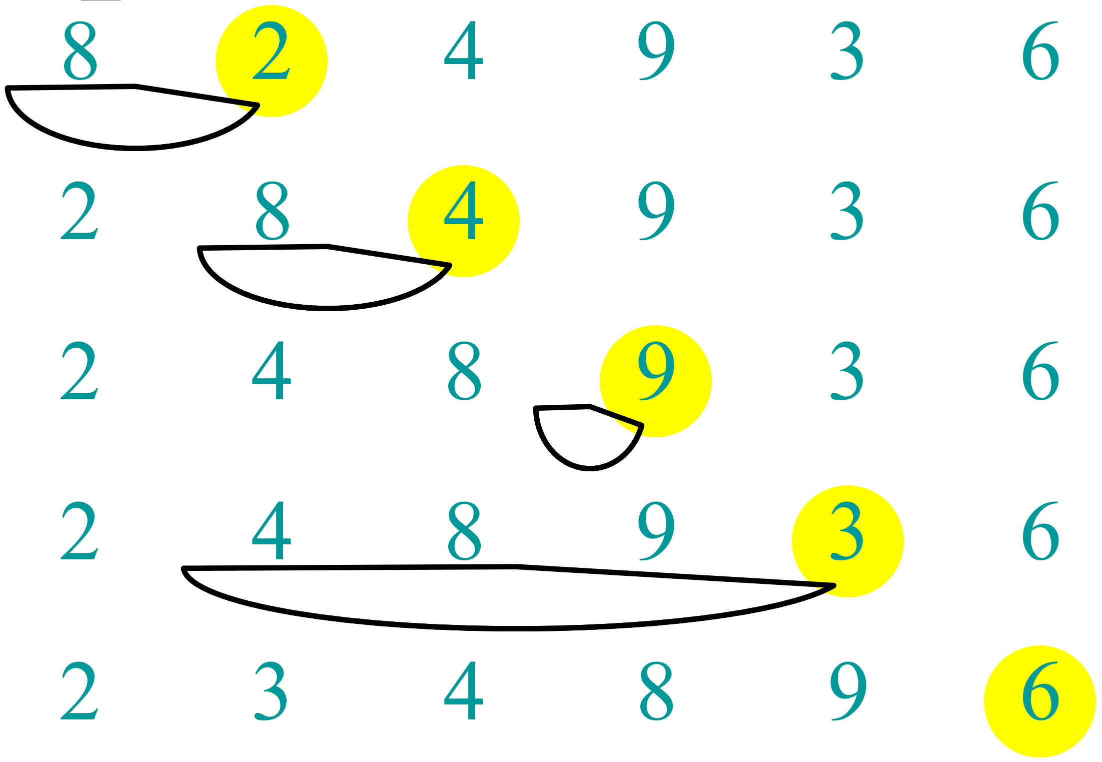
Example of insertion sort



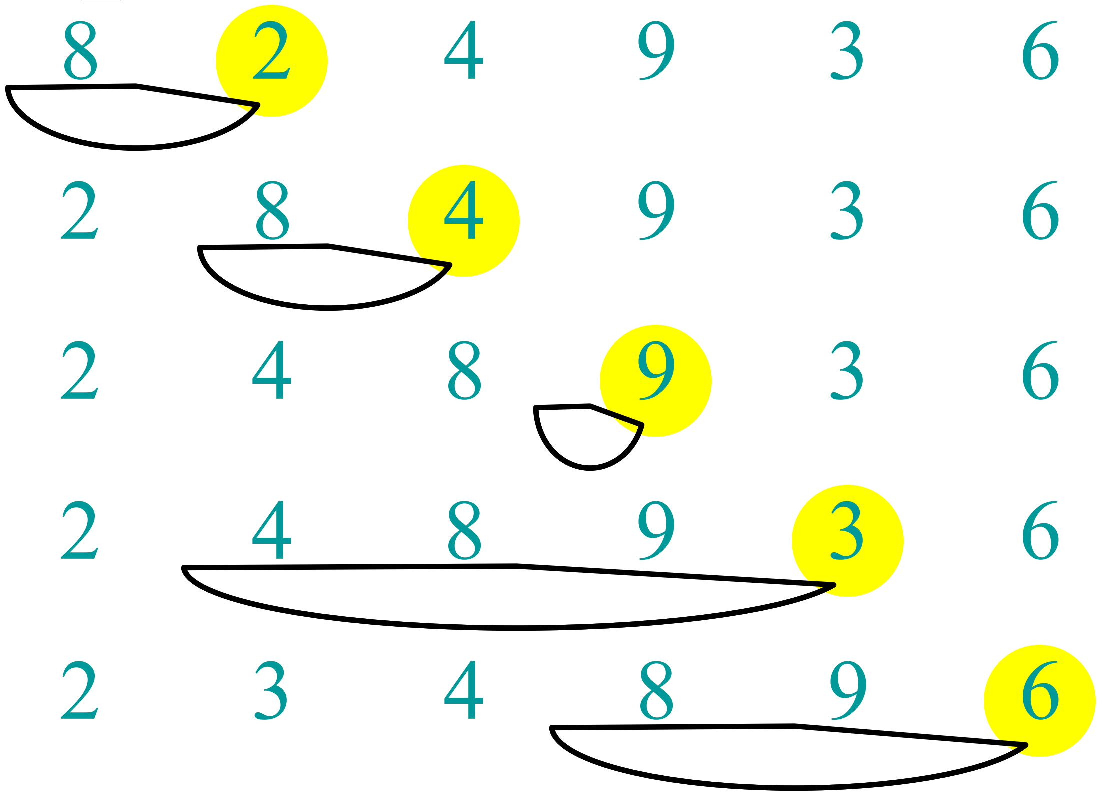
Example of insertion sort



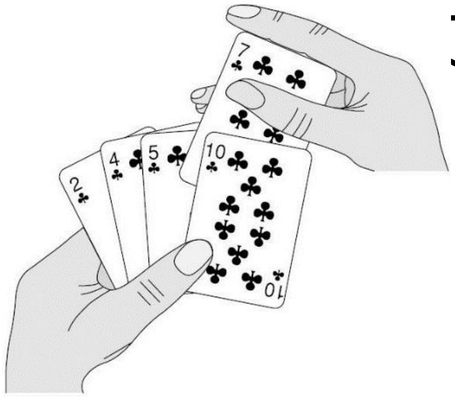
Example of insertion sort



Example of insertion sort



Example of insertion sort



8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

2 3 4 6 8 9 *done*

C++ Implementation of Insertion Sort

```
int main(){
    int arr[] = {10, 6, 3, 2, 1, 8};
    int l = sizeof(arr)/sizeof(*arr);
    print(arr, l);
    insertionSort(arr, l);
    print(arr, l);

}
```

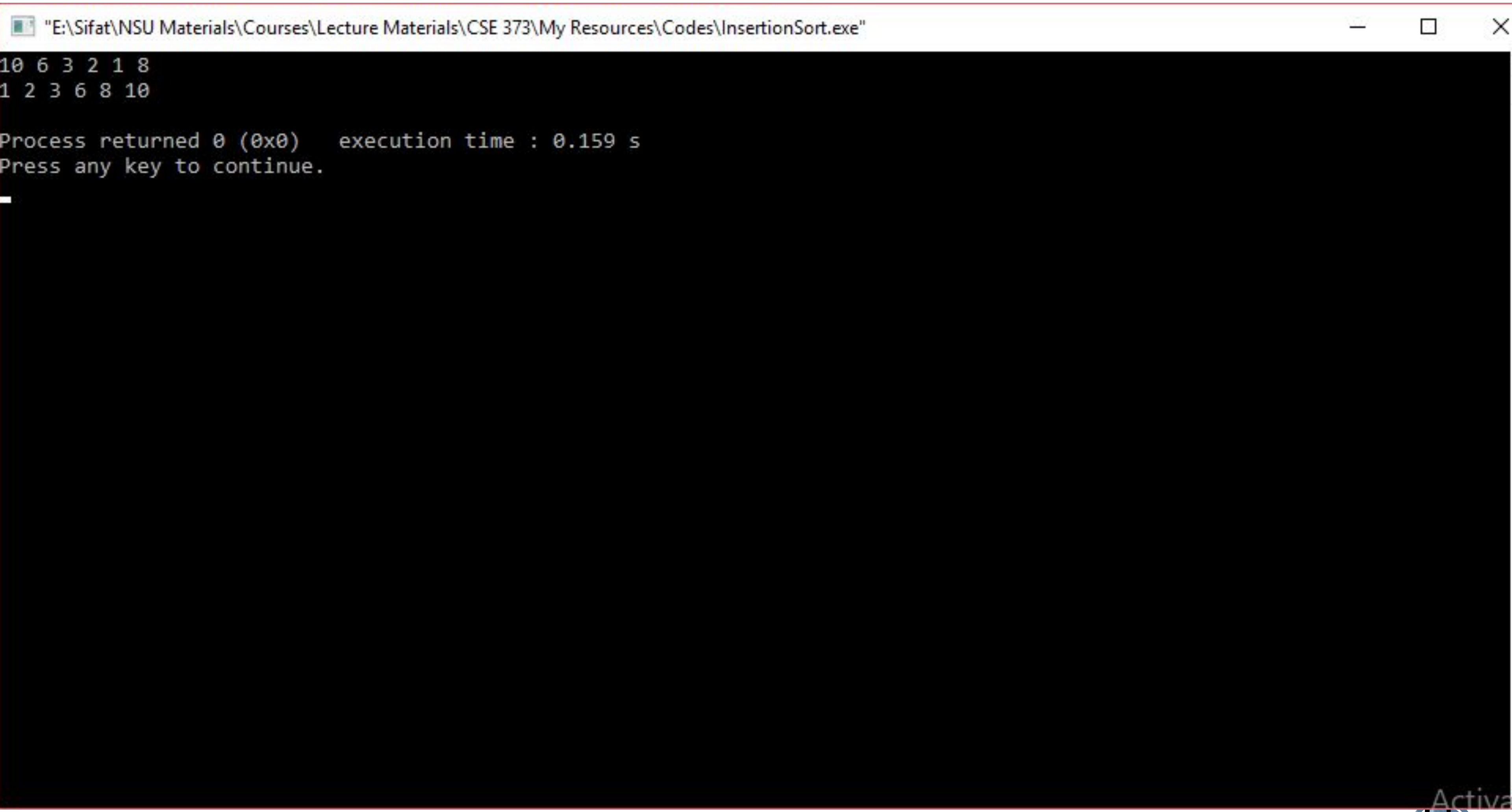
C++ Implementation of Insertion Sort

```
void insertionSort(int A[], int length){
    int key, i;
    for(int j = 1; j < length; j++){
        key = A[j];
        i = j - 1;
        while(i > -1 && A[i] > key){
            A[i+1] = A[i];
            i = i - 1;
        }
        A[i+1] = key;
    }
}
```

Print Function

```
void print(int a[], int length){
    for(int i = 0; i < length; i++)
        cout << a[i] <<" ";
    cout <<endl;
}
```

Output


"E:\Sifat\NSU Materials\Courses\Lecture Materials\CSE 373\My Resources\Codes\InsertionSort.exe"

```
10 6 3 2 1 8
1 2 3 6 8 10

Process returned 0 (0x0)   execution time : 0.159 s
Press any key to continue.
```

Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Major Simplifying Convention: Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.

  $T_A(n) =$ time of A on length n inputs
- Generally, we seek upper bounds on the running time, to have a guarantee of performance.

Kinds of analyses

Worst-case: (usually)

- $T(n)$ = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case: (NEVER)

- Cheat with a slow algorithm that works fast on *some* input.

Insertion Sort

```
INSERTION-SORT $(A, n)$ ▷ $A[1 \, . \, . \, n]$
  for    $j \leftarrow 2$ to $n$
     do  $key \leftarrow A[j]$
         $i \leftarrow j - 1$
         while   $i > 0$ and $A[i] > key$
              do  $A[i+1] \leftarrow A[i]$
                  $i \leftarrow i - 1$
         $A[i+1] = key$
```

What is the estimated running time?
Depends on arrangement of numbers in the input array. ***We are typically interested in the runtime of an algorithm in the worst case scenario***.
Because it provides us a guarantee that the algorithm won't take any longer than this for any type of input.

How can you arrange the input numbers so that this algorithm becomes most inefficient (worst case)?

Analyzing Algorithms

- Analyzing an algorithm means predicting the resources that the algorithm requires.
  - Typically time and memory
- Before we can analyze an algorithm, we need a model of implementation technologies that we will use.
  - Random Access Machine Model

Random Access Machine model

- Single Processor
- Instructions are executed one after another, with no concurrent operations
- The following instructions take a constant amount of time
  - Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling etc…
  - Data movement: load, store, copy
  - Control: conditional/unconditional branch, subroutine call, return
  - Data type: Integer and Float

Insertion Sort: Running Time

Statement	cost
INSERTION-SORT (A, n) ▷ $A[1\,..\,n]$	
for $j \leftarrow 2$ **to** n	$c_1 n$
do $key \leftarrow A[j]$	$c_2(n-1)$
$i \leftarrow j-1$	$c_3(n-1)$
while $i > 0$ and $A[i] > key$	$c_4 \sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	$c_5 \sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	$c_6 \sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	$c_7(n-1)$

$$T(n)$$

$$= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Here t_j = no. of times the condition of while loop is tested for the current value of j.
In the **worst case** (when input is reverse-sorted), in each iteration of the for loop, all the *j-1* elements need to be right shifted and the key will be inserted in the front of them, *i.e.,* $t_j = j$. *Us*ing this in the above equation, we get: T(n) = An^2+Bn+C, where A, B, C are constants.

What is T(n) in the best case (when the input numbers are already sorted)?

Insertion Sort: Running Time (Best case)

- The best case is when the input is already in the sorted manner.
- Thus t_j = *1*

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

This can be expressed as $T(n) = an + b$, thus $T(n) = O(n)$

Insertion Sort: Running Time (Worst case)

- The worst case results when the array is in the reverse order (in this case decreasing order)
- In this situation, we must compare each element A[j] with each element in the entire sorted sub-array A[1… j-1]
  - This results $t_j = j$

$$\sum_{j=2}^{n} (j - 1) = \frac{n(n-1)}{2}$$

Insertion Sort: Running Time (Worst case)

- Thus

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

This can be expressed as $T(n) = an^2 + bn + c$, thus $T(n) = O(n^2)$

Asymptotic Analysis

To compare two algorithms with running times *f(n)* and *g(n)*, we need a **rough measure** that characterizes **how fast each function grows.**

Hint: use *rate of growth*

Compare functions in the limit, that is, **asymptotically!**

(i.e., for large values of *n*)

Rate of Growth

Consider the example of buying *elephants* and *goldfish:*

Cost: cost_of_elephants + cost_of_goldfish

Cost ~ cost_of_elephants (approximation)

The low order terms, as well as constants in a function are relatively insignificant for **large** *n*

$$6n + 4 \sim n$$

$$n^4 + 100n^2 + 10n + 50 \sim n^4$$

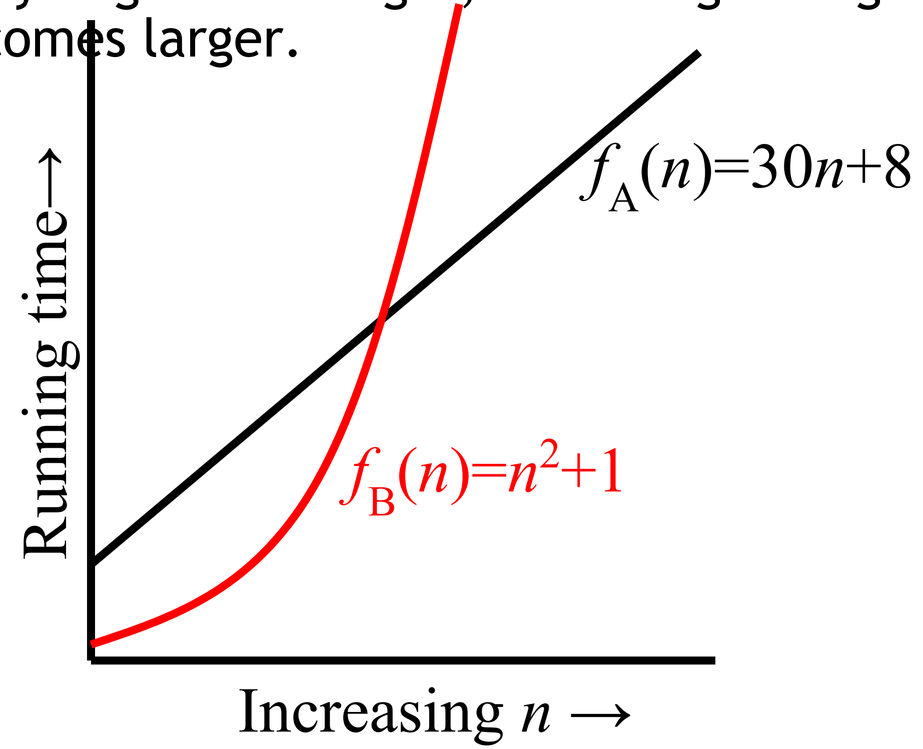
i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**

Big-O Notation

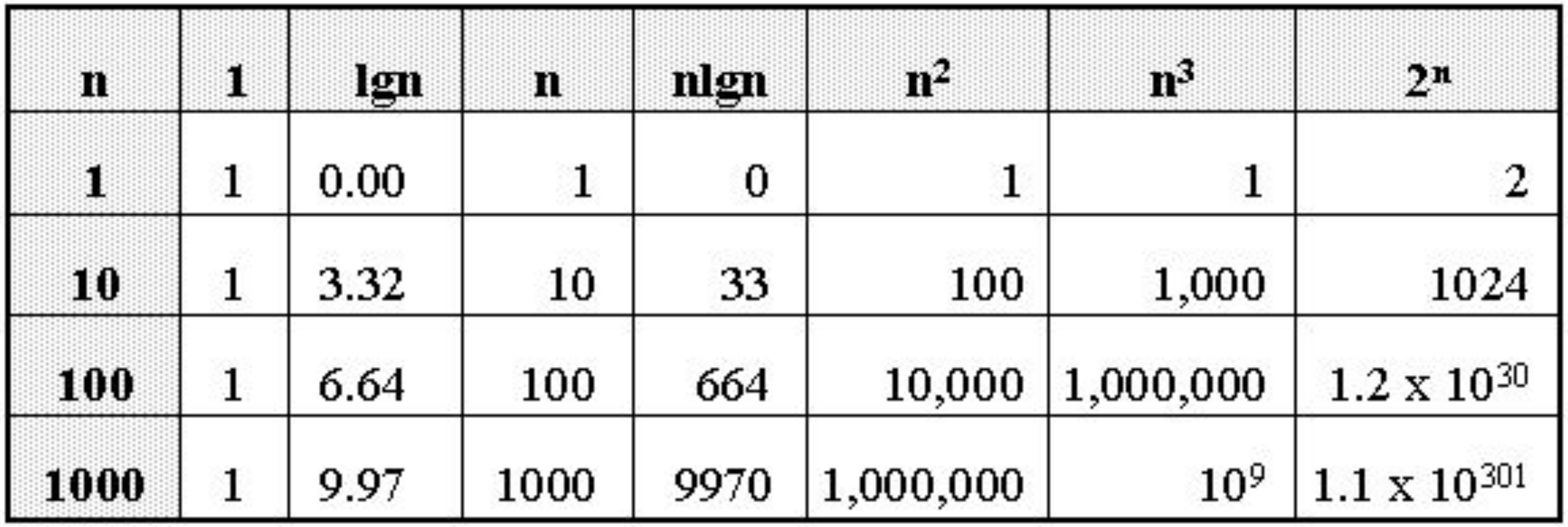
- We say $f(n) = 30000$ is in the *order of* 1, or $\boldsymbol{O(1)}$
    - Growth rate of 30000 is constant, that is, it is not dependent on problem size.
- $f(n) = 30n + 8$ is in the *order of* n, or $\boldsymbol{O(n)}$
    - Growth rate of $30n + 8$ is roughly *proportional* to the growth rate of n.
- $f(n) = n^2 + 1$ is in the *order of* n^2, or $\boldsymbol{O(n^2)}$
    - Growth rate of $n^2 + 1$ is roughly proportional to the growth rate of n^2.
- In general, any $O(n^2)$ function is faster- growing than any $O(n)$ function.
    - For large n, a $O(n^2)$ algorithm runs a lot slower than a $O(n)$ algorithm.

Visualizing Orders of Growth

On a graph, as you go to the right, a faster growing function <u>eventually</u> becomes larger.

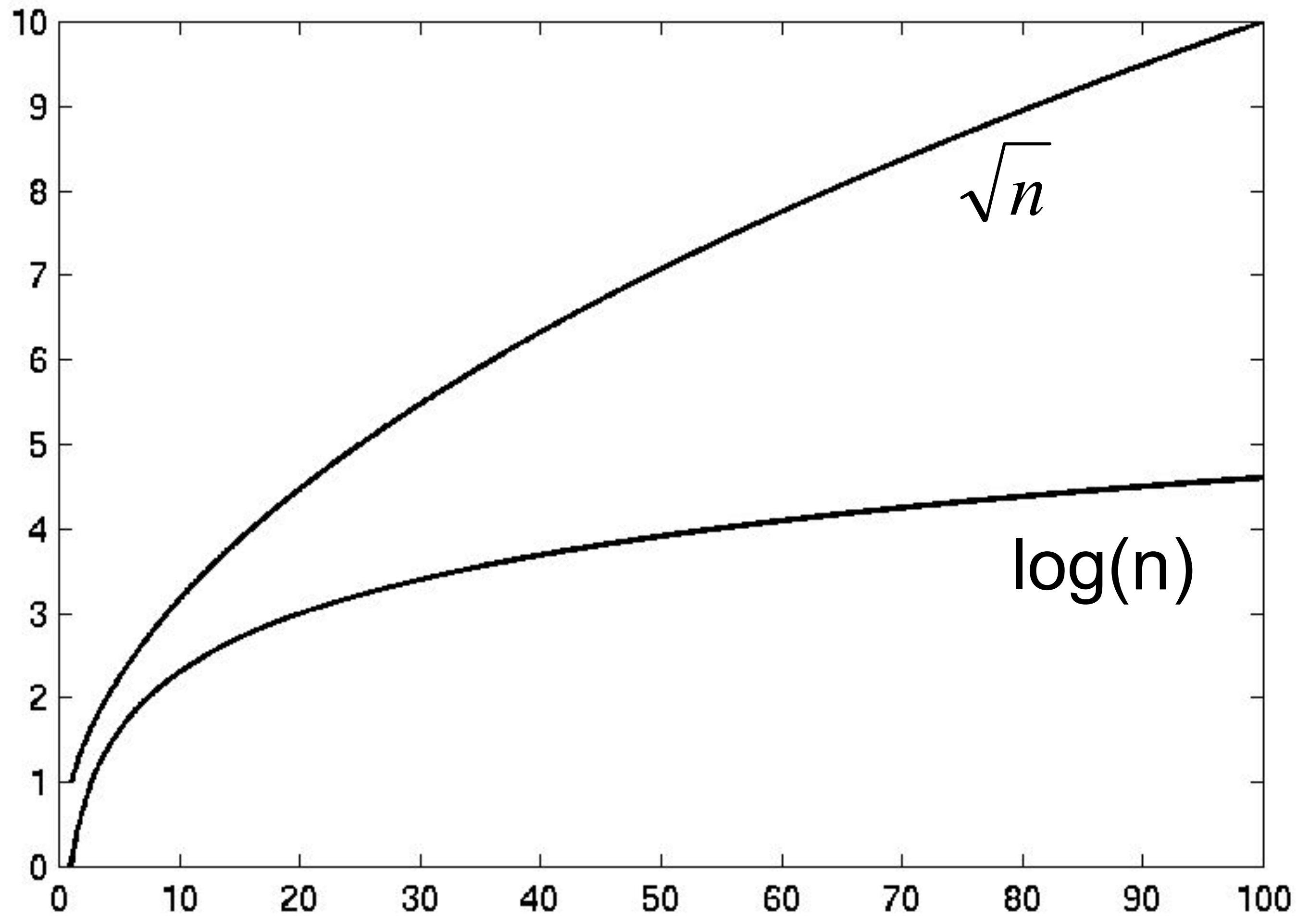


Growth of Functions

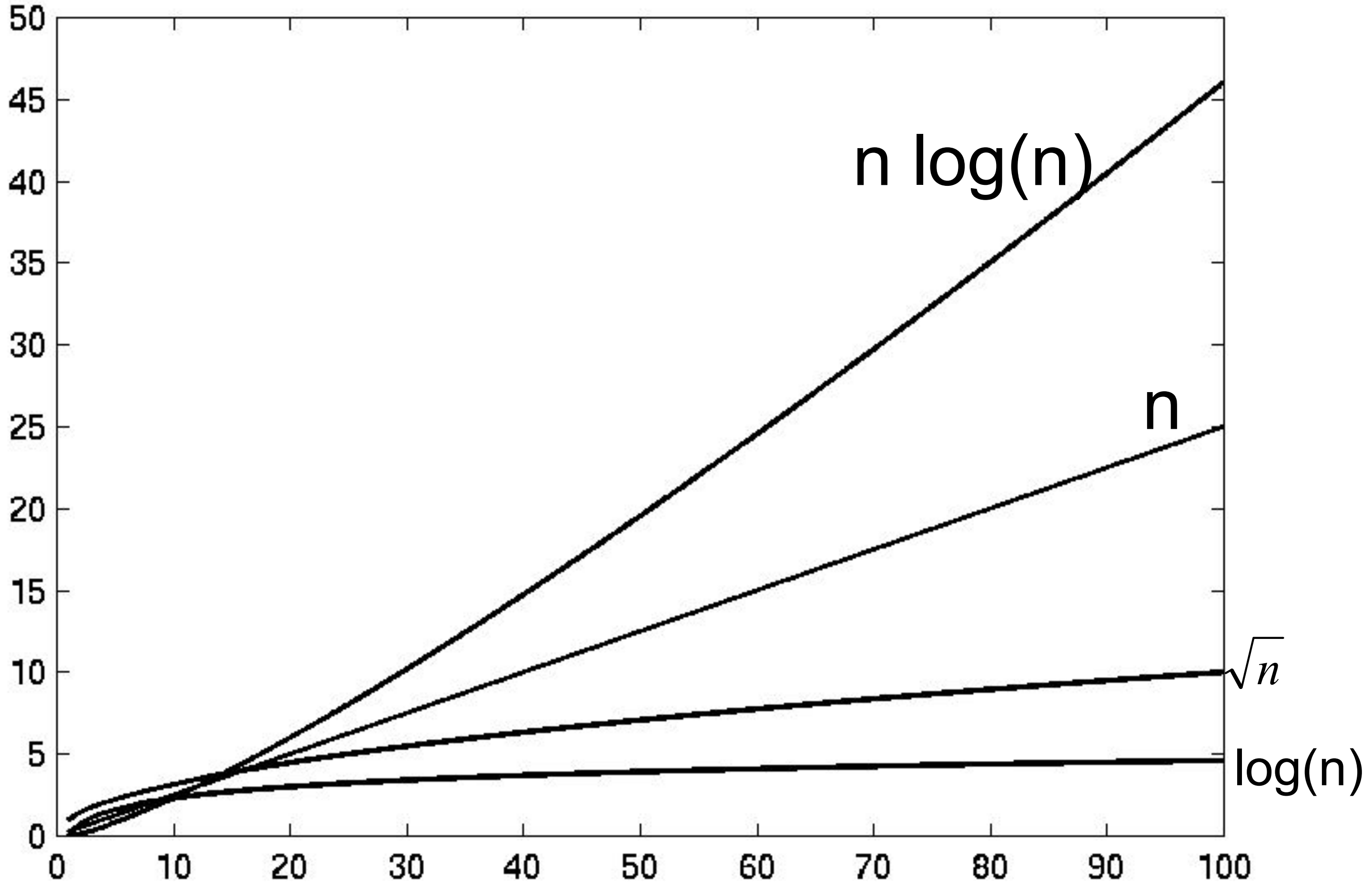


n	1	lgn	n	nlgn	n^2	n^3	2^n
1	1	0.00	1	0	1	1	2
10	1	3.32	10	33	100	1,000	1024
100	1	6.64	100	664	10,000	1,000,000	1.2×10^{30}
1000	1	9.97	1000	9970	1,000,000	10^9	1.1×10^{301}

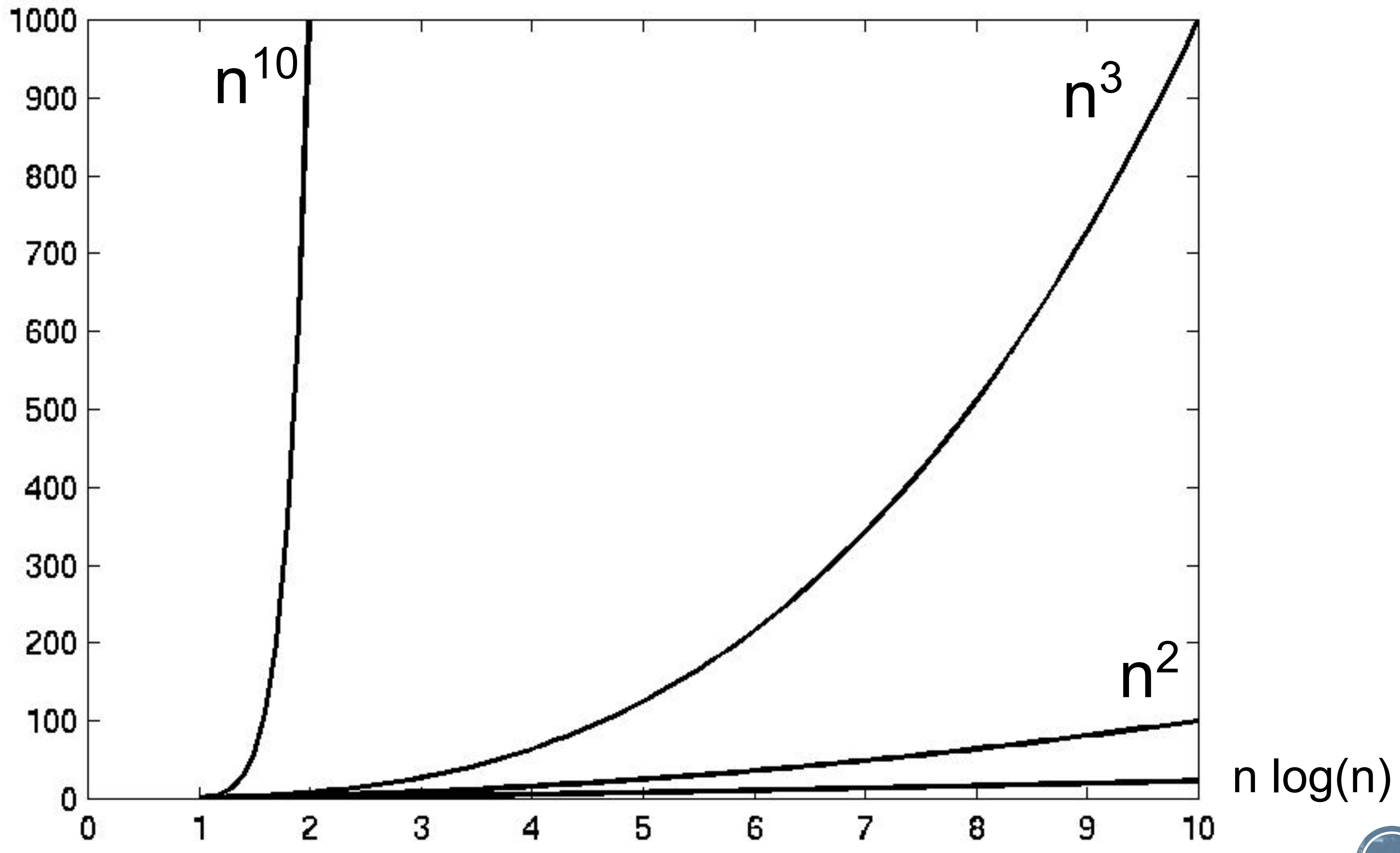
Complexity Graphs



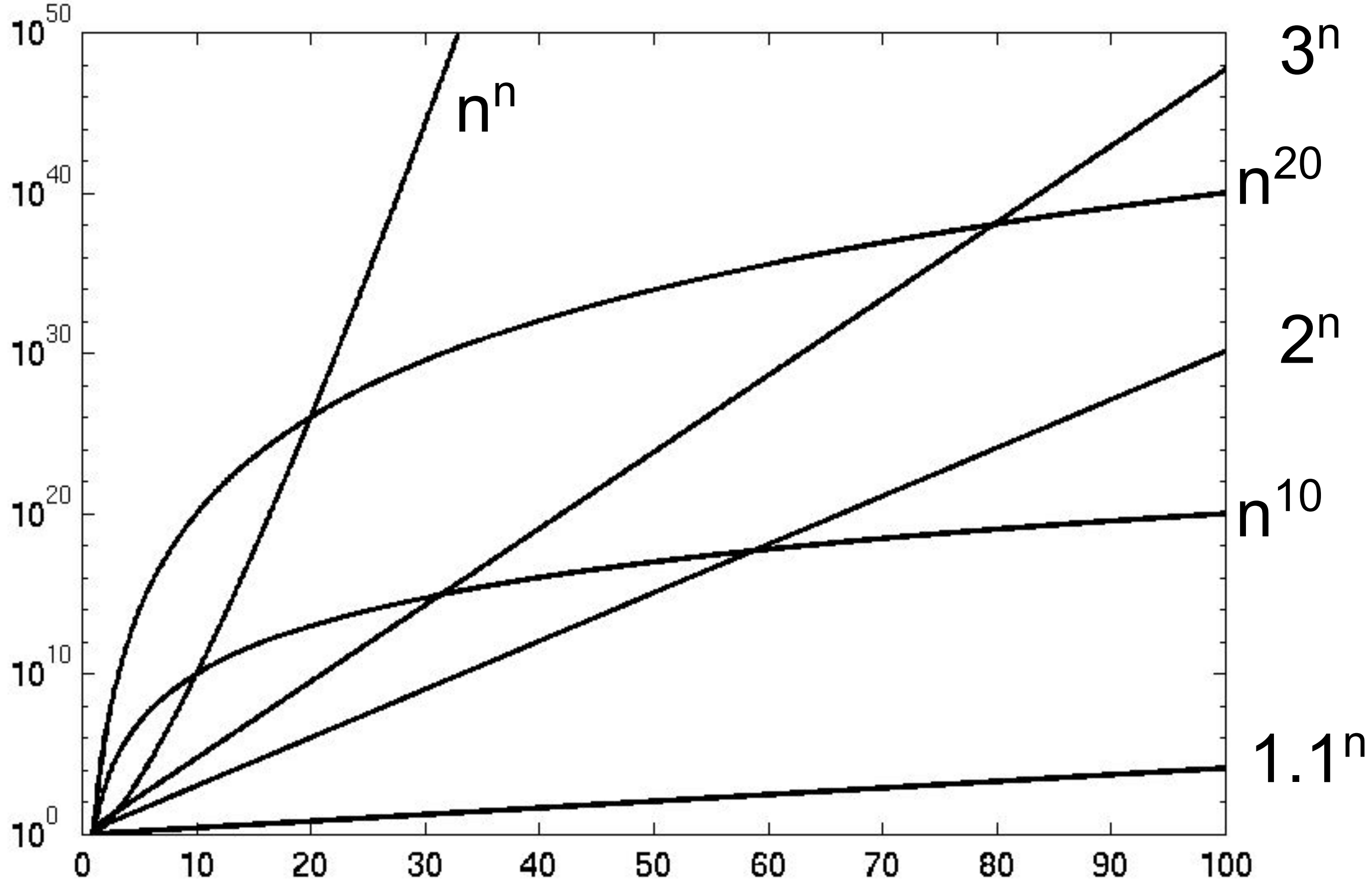
Complexity Graphs



Complexity Graphs



Complexity Graphs (log scale)



Asymptotic Notations

O notation: asymptotic “upper bound”:

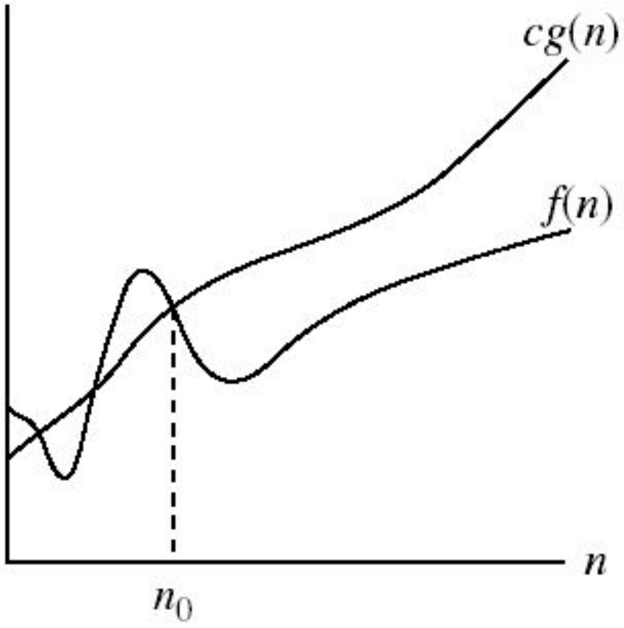
Ω notation: asymptotic “lower bound”:

Θ notation: asymptotic “tight bound”:

Asymptotic Notations

- *O-notation*

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



$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$

Examples:

$T(n) = 3n^2+10n\lg n+8$ is $O(n^2)$, $O(n^2\lg n)$, $O(n^3)$, $O(n^4)$, …

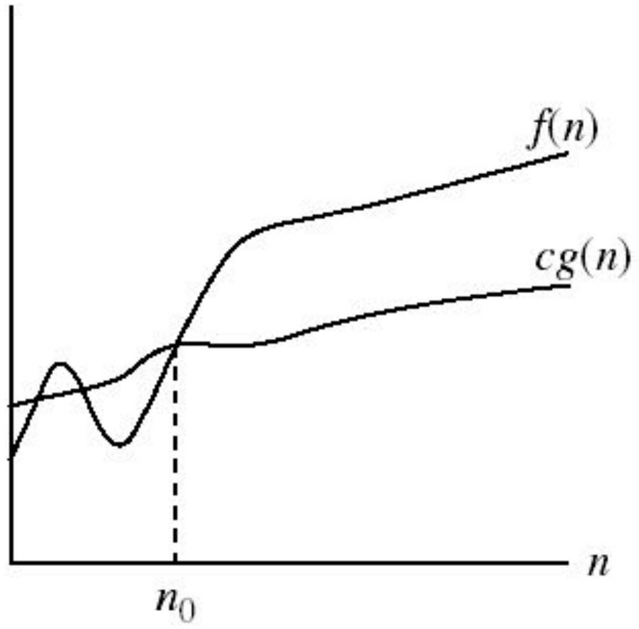
$T'(n) = 52n^2+3n^2\lg n+8$ is $O(n^2\lg n)$, $O(n^3)$, $O(n^4)$, …

$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Asymptotic Notations

- *Ω - notation*

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



$\Omega(g(n))$ is the set of functions with larger or same order of growth as $g(n)$

Examples:

$T(n)=3n^2+10n\lg n+8$ is $\Omega(n^2)$, $\Omega(n\lg n)$, $\Omega(n)$, $\Omega(n\lg n)$,$\Omega(1)$

$T'(n) = 52n^2+3n^2\lg n+8$ is $\Omega(n^2\lg n)$, $\Omega(n^2)$, $\Omega(n)$, …

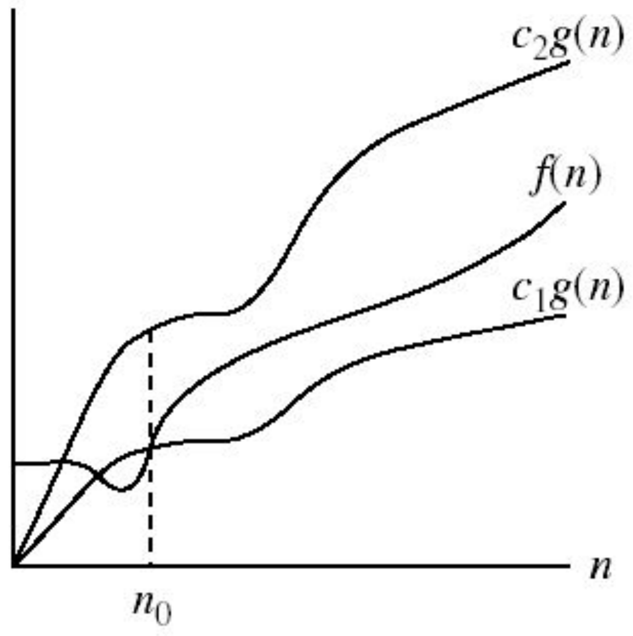
$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Asymptotic Notations

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$

$\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$

* $f(n)$ is both $O(g(n))$ & $\Omega(g(n))$ $\leftrightarrow$ $f(n)$ is $\Theta(g(n))$



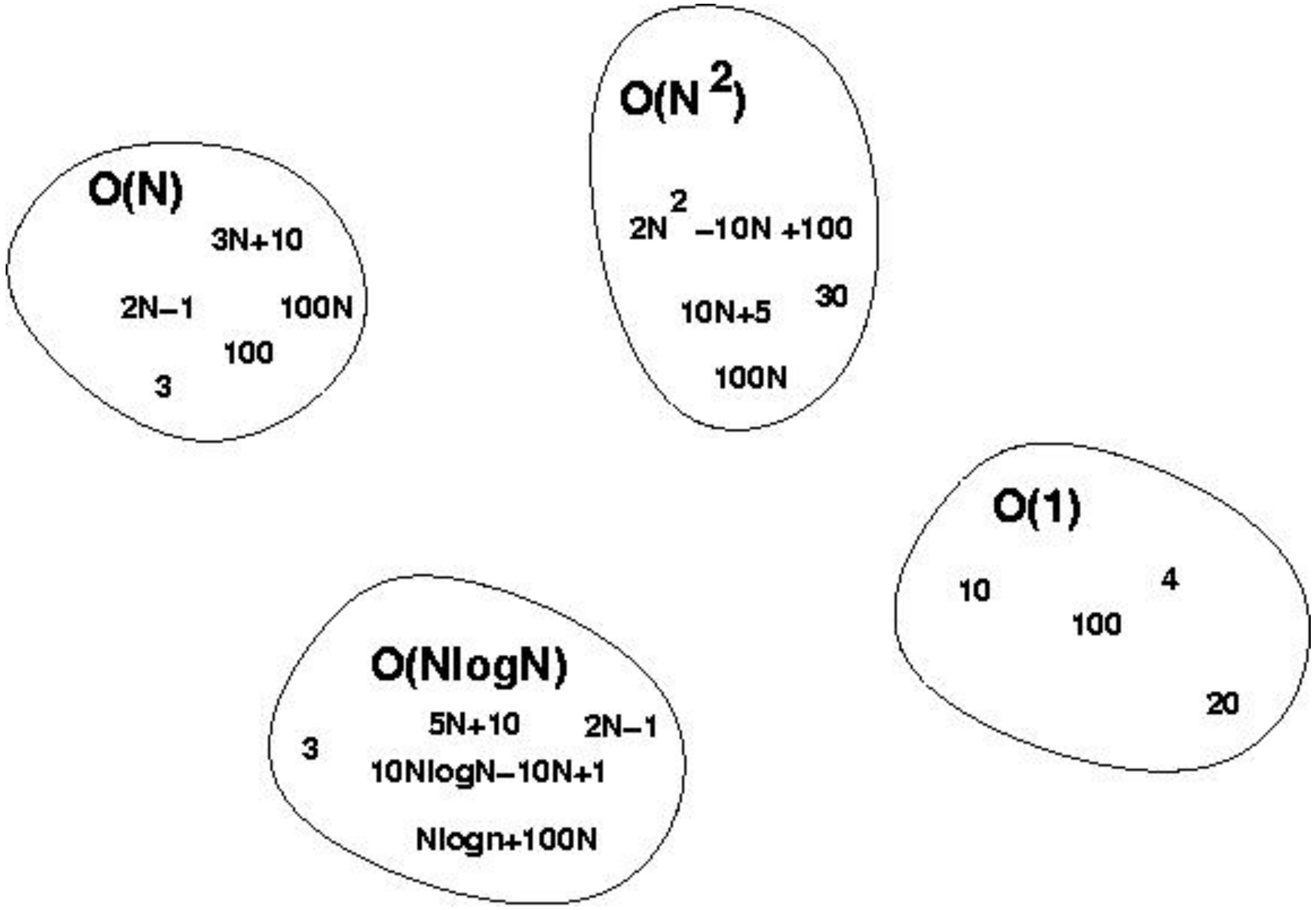
Examples:

$T(n) = 3n^2+10n\lg n+8$ is $\Theta(n^2)$

$T'(n) = 52n^2+3n^2\lg n+8$ is $\Theta(n^2\lg n)$

$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Big-O Visualization



Some Examples

Determine the time complexity for the following algorithm.

```
count = 0;
for(i=0; i<10000; i++)
     count++;
```

Some Examples

Determine the time complexity for the following algorithm.

$\boldsymbol{O}(\mathbf{1})$

```
count = 0;
for(i=0; i<10000; i++)
     count++;
```

Some Examples

Determine the time complexity for the following algorithm.

```
count = 0;
for(i=0; i<n; i++)
     count++;
```

Some Examples

Determine the time complexity for the following algorithm.

$O(n)$

```
count = 0;
for(i=0; i<n; i++)
     count++;
```

Some Examples

Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=0; i<n; i++)
     for(j=0; j<n; j++)
        sum += arr[i][j];
```

Some Examples

Determine the time complexity for the following algorithm.

$O(n^2)$

```
sum = 0;
for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            sum += arr[i][j];
```

Some Examples

Determine the time complexity for the following algorithm.

```
count = 0;
for(i=1; i<=n; i=i*2)
      count++;
```

Some Examples

Determine the time complexity for the following algorithm.

```
count = 0;
for(i=1; i<=n; i=i*2)
    count++;
```

***O*(lg n)**

Some Examples

Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=1; i<n; i=i*2)
     for(j=0; j<n; j++)
        sum += i*j;
```

Some Examples

Determine the time complexity for the following algorithm. ***O*(n lg n)**

```
sum = 0;
for(i=1; i<n; i=i*2)
     for(j=0; j<n; j++)
        sum += i*j;
```

Some Examples

Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=1; i<=n; i=i*4)
     for(j=0; j<=n; j*=2)
          sum += i*j;
```

Asympotic Tight Bound: *Θ(lg n)* **WHY?**

Some Examples

Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=1; i<n; i=i*2)
     for(j=0; j<i; j++)
          sum += i*j;
```

Some Examples

Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=1; i<n; i=i*2)
     for(j=0; j<i; j++)
          sum += i*j;
```

Loose Upper Bound: ***$O(n \lg n)$***

Tight Upper Bound: ***$O(n)$***

Asympotic Tight Bound: ***$\Theta(n)$***

WHY?

Some Examples

Determine the time complexity for the following algorithm.

```
char someString[10];
gets(someString);
for(i=0; i<strlen(someString); i++)
     someString[i] -= 32;
```

Some Examples

Determine the time complexity for the following algorithm.

```
char someString[10];
gets(someString);
for(i=0; i<strlen(someString); i++)
     someString[i] -= 32;
```

$\boldsymbol{O(n^2}$

Types of Analysis

- Is input size everything that matters?

```
int find_a(char *str)
{
    int i;
    for (i = 0; str[i]; i++)
    {
        if (str[i] == ’a’)
            return i;
    }
    return -1;
}
```

- **Time complexity:** $O(n)$

- Consider two inputs: “alibi” and “never”