Course Code

CSC402

Course Name

Analysis of Algorithms

Department of Computer Engineering

AY 2021-2022

Course Objectives

1. To provide mathematical approaches for Analysis of Algorithms
2. To understand and solve problems using various algorithmic approaches
3. To analyze algorithms using various methods

Course Outcome

1. Analyze the running time and space complexity of algorithms.
2. Describe, apply and analyze the complexity of divide and conquer strategy.
3. Describe, apply and analyze the complexity of greedy strategy.
4. Describe, apply and analyze the complexity of dynamic programming strategy.
5. Explain and apply backtracking, branch and bound.
6. Explain and apply string matching techniques.

Syllabus

Module 1 Introduction

- Performance analysis, space, and time complexity Growth of function, Big-Oh, Omega Theta notation Mathematical background for algorithm analysis.
- Complexity class: Definition of P, NP, NP-Hard, NP-Complete Analysis of selection sort, insertion sort.
- Recurrences: The substitution method, Recursion tree method, Master method

Module 2 Divide and Conquer Approach

- General method, Merge sort, Quick sort, Finding minimum and maximum algorithms and their Analysis, Analysis of Binary search.

Syllabus

Module 3 Greedy Method Approach

- General Method, Single source shortest path: Dijkstra Algorithm Fractional Knapsack problem, Job sequencing with deadlines, Minimum cost spanning trees: Kruskal and Prim's algorithms

Module 4 Dynamic Programming Approach

- General Method, Multistage graphs, Single source shortest path: Bellman Ford Algorithm All pair shortest path: Floyd Warshall Algorithm, Assembly-line scheduling Problem0/1 knapsack Problem, Travelling Salesperson problem, Longest common subsequence

Syllabus

Module 5 Backtracking and Branch and bound

- General Method, Backtracking: N-queen problem, Sum of subsets, Graph coloring
- Branch and Bound: Travelling Salesperson Problem, 15 Puzzle problem

Module 6 String Matching Algorithms

- The Naïve string-matching algorithm, The Rabin Karp algorithm, The Knuth-Morris-Pratt algorithm

books

Textbooks

- T. H. Cormen, C.E. Leiserson, R. L. Rivest, and C. Stein, “Introduction to algorithms”, 2nd Edition, PHI Publication 2005.
- Ellis Horowitz, Sartaj Sahni, S. Rajsekaran. “Fundamentals of computer algorithms” University Press

References:

- Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani, “Algorithms”, Tata McGrawHill Edition.
- S. K. Basu, “Design Methods and Analysis of Algorithm”, PHI

Module 1 Introduction

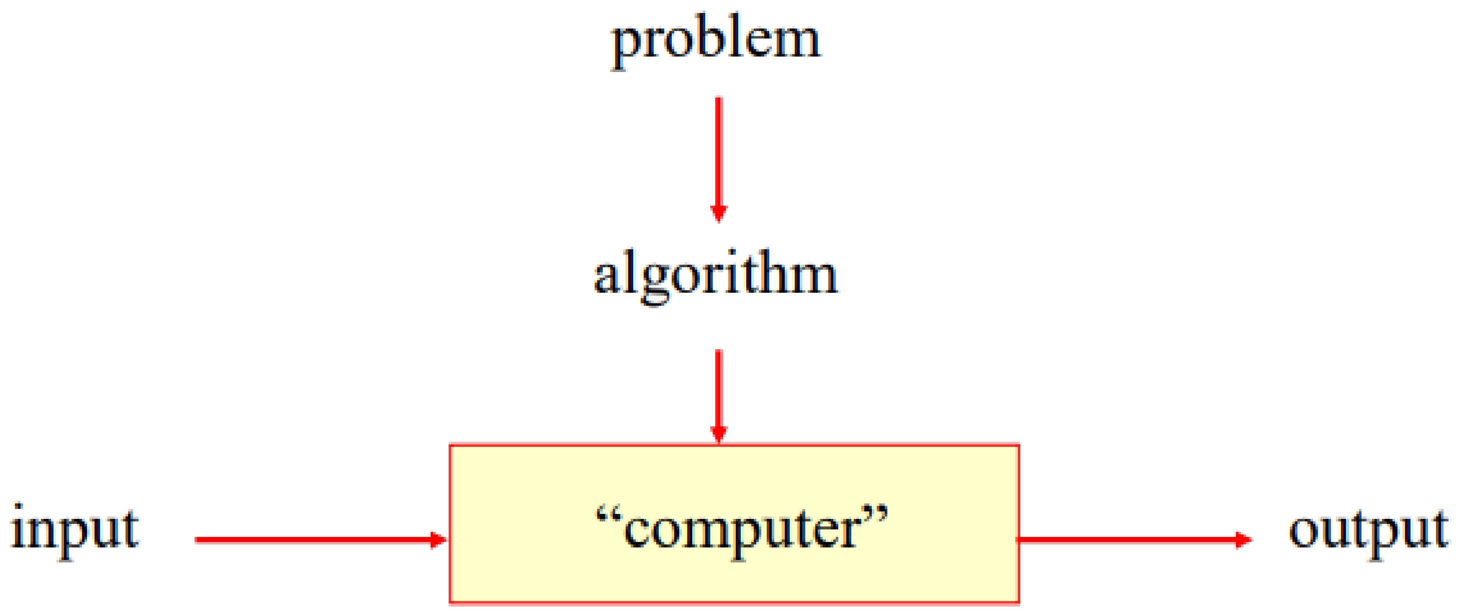
CE– SE–AOA

Dr. Anil Kale

Associate Professor

Dept. of Computer Engineering,

Notion of algorithm



Algorithmic solution

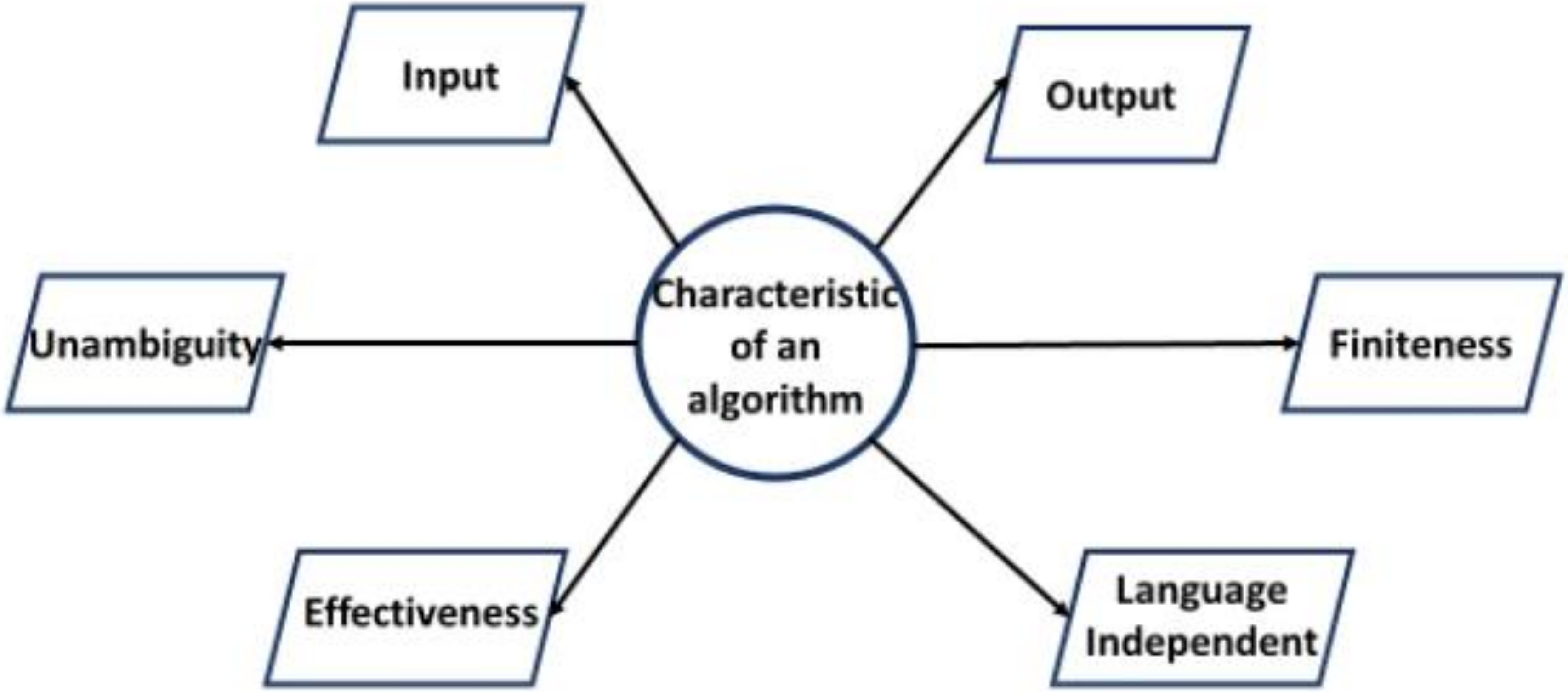
Definitions of Algorithm

- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.
- A mathematical relation between an observed quantity and a variable used in a step-by-step mathematical process to calculate a quantity
- Algorithm is any well defined computational procedure that takes some value or set of values as input and produces some value or set of values as output
- A procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation; broadly : a step-by-step procedure for solving a problem or accomplishing some end (Webster's Dictionary)

What is the difference between program and algorithm?

Algorithm	Program
Design	Implementation
domain Knowledge	Programmer
Any language means English language	Programming language
Hardware and Software men's Operating System Independent	Dependent on Hardware and Operating System
Analyze	Testing

What are the characteristic of Algorithm



What are the characteristic of Algorithm

Recipe, process, method, technique, procedure, routine,… with following requirements:

1. Finiteness
   - ➡ terminates after a finite number of steps
2. Definiteness
   - ➡ rigorously and unambiguously specified
3. Input
   - ➡ valid inputs are clearly specified
4. Output
   - ➡ can be proved to produce the correct output given a valid input
5. Effectiveness
   - ➡ steps are sufficiently simple and basic

How to write an Algorithm

- There are no well-defined standards for writing algorithms. It is, however, a problem that is resource-dependent. Algorithms are never written with a specific programming language in mind.
- As you all know, basic code constructs such as loops like do, for, while, all programming languages share flow control such as if-else, and so on. An algorithm can be written using these common constructs.
- Algorithms are typically written in a step-by-step fashion, but this is not always the case. Algorithm writing is a process that occurs after the problem domain has been well-defined. That is, you must be aware of the problem domain for which you are developing a solution.

```
Algorithm swap(a,b)
{
    Temp = a;
    A=b;
    B=temp;
}
```

How to Analyze an Algorithm

Algorithm analysis is defined as determining the efficiency and quality of the algorithm and then developing it better. The extent and quality of the algorithm are measured by two measures:

- **Space Complexity:** The amount of memory needed by the program (from its operation to completion).
- **Time complexity:** It is the amount of time needed to form and configure a program until it is finished.

Analyzing means predicting resources that algorithms requires , i.e.

- Space complexity
  - How much space is required
- Time complexity
  - How much time does it take to run the algorithm Often, we deal with estimates!

Frequency Count Method

1)

```
Algorithm sum(A,n)
{
        S=0;
        For(i=0; i<n; i++)
        {
        S=s+A[i];
        }
Return s;
}
```

Frequency Count Method

2)

Algorithm add(A,B,n)

```
{
        For(i=0; I<n; i++)
        For(j=0; j<n; j++)
        {
             C[I,j]=A[I,j]+B[I,j];
        }
}
```

Frequency Count Method

```
3)
Algorithm Mult(A,B,n)
{
     For(i=0; i<n; i++)
        For(j=0; j<n; j++)
        {
          C[i,j]=0;
          {
          For(k=0; k<n; k++)
            {
            C[i,j]=C[i,j]+A[i,k]+B[k,j];
            }
         }
}
```

Frequency Count Method

<u>1)</u>

```
For(i=0; i<n; i++)
    {
    statement;
    }
```

2)

```
For(i=n; i>0; i--)
    {
    statement;
    }
```

3)

```
For(i=0; i<n; i++)
    For(j=0; j<n; j++)
           {
           statement;
           }
```

Frequency Count Method

4)

```
For(i=0; i<n; i++)
     For(j=0; j<i; j++)
          {
          statement;
          }
```

Frequency Count Method

<u>5)</u>

```
P=0;
For(i=1; P<=n; i++)
    {
    P=P+i;
    }
```

Frequency Count Method

6)

```
For(i=1; i<n; i=i*2)
     {
      statement;
     }
```

Frequency Count Method

7)

```
For(i=n; i>=1; i=i/2)
    {
    statement;
    }
```

Frequency Count Method

8)

```
For(i=0; i<n; i++)
    For(j=0; j<n; j=j*2)
           {
           statement;
           }
```

Frequency Count Method

```
For(i=0; i<n; i++)
{
    statement;
}
For(j=0; j<i; j++)
{
    statement;
}
```

Frequency Count Method

```
P=0;
For(i=1; i<n; i=i*2)
{
    P=P+i;
}
For(j=1; j<P; j=j*2)
{
    statement;
}
```

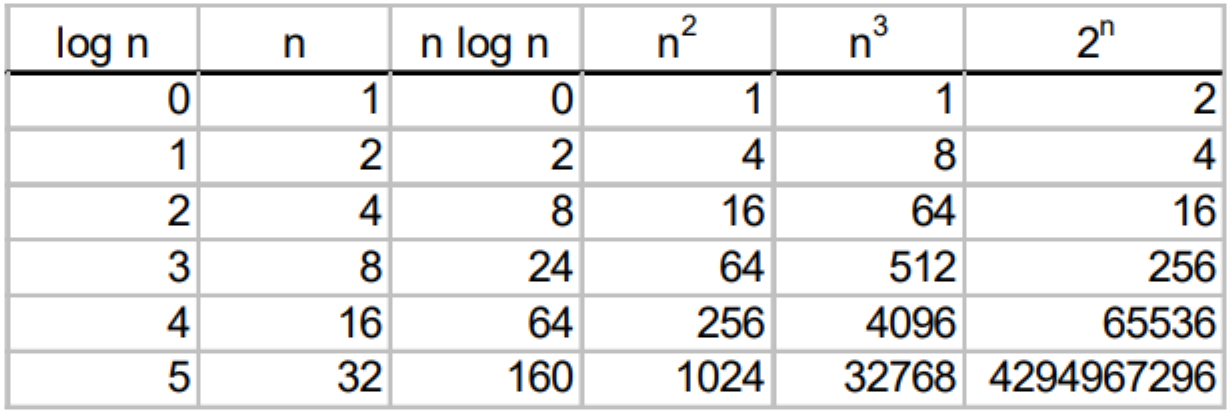
Frequency Count Method

```
For(i=0; i<n; i++)
{
    For(j=1; j<n; j=j/2)
    {
        statement;
    }
}
```

Types of function

- O(1): constant
- O(log2(n)): logarithmic
- O(nlog(n)) : logarithmic
- O(n): linear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential

$O(1)$, $O(\log_2 n)$, $O(n)$, $O(n. \log_2 n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, n! and n^n

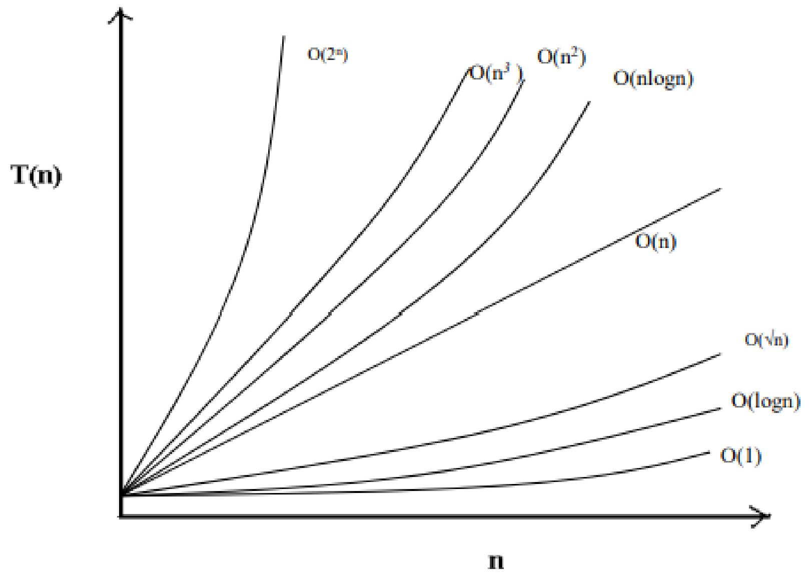


log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Types of function

O(1), $O(\log_2 n)$, O(n), $O(n.\log_2 n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, n! and n^n

log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296



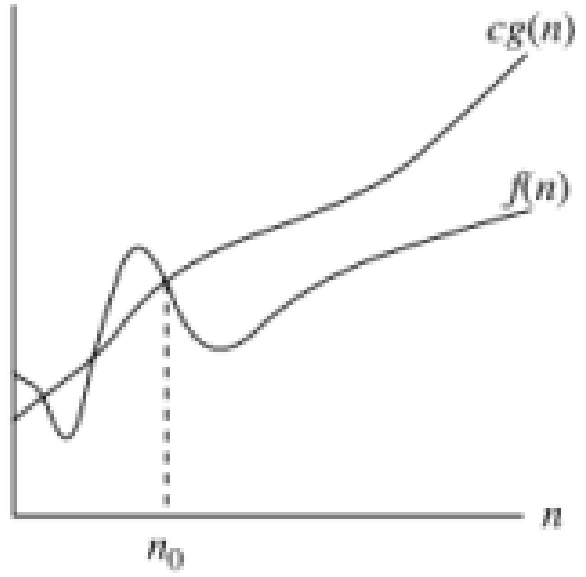
ASYMPTOTICS NOTATIONS

- O-notation (Big Oh) - Upper Bound
- Ω-notation (Big–Omega) - lower bound
- Θ-notation (Big–Theta) - tight bound

ASYMPTOTICS NOTATIONS

- O-notation (Big Oh) - Upper Bound

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

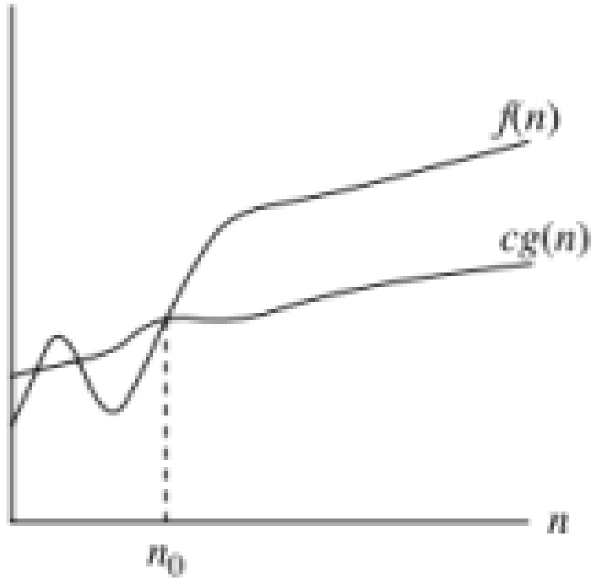


$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

ASYMPTOTICS NOTATIONS

- Ω-notation (Big–Omega) - lower bound

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

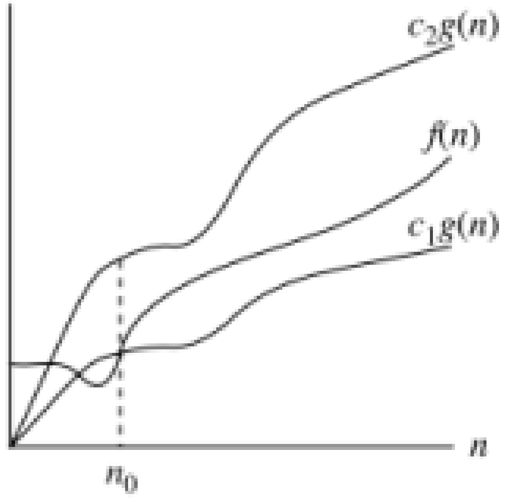


$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

ASYMPTOTICS NOTATIONS

- Θ-notation (Big–Theta) - tight bound

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



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

Best, Worst, Average Case Analysis

- **Best case Running Time**: Searching key element present at first index.
- **Worst case Running Time**: Searching key element present at last index.
- **Average case Running Time**: All passible case time / number of cases

Recurrences

- Recursion is generally expressed in terms of recurrences. In other words, when an algorithm calls to itself, we can often describe its running time by a recurrence equation which describes the overall running time of a problem of size n in terms of the running time on smaller inputs.
- **There are various techniques to solve recurrences.**
  - Substitution Method:
  - Recursion Tree Method:
  - Master Method:

Recurrences

```
Void Test(int n)
{
    if(n>0)
     printf('%d', n);
     Test(n-1)
}
```

Recurrences

```
Void Test(int n)
{
    if(n>0)
     {
      for (i=0; i<n; i++)
         {
           printf('%d', n);
         }
     }
     Test(n-1)
}
```

Recurrences

```
Void Test(int n)
{
    if(n>0)
     {
      for (i=1; i<n; i=i*2)
         {
           printf('%d', i);
         }
     }
     Test(n-1)
}
```

Recurrences

```
Void Test(int n)
{
    if(n>0)
     printf('%d', n);
     Test(n-1);
     Test(n-1);
}
```

Recurrences

Master theorem for Decreasing function

General form for decreasing functions will be

- **T(n) = aT(n-b) + f(n)**, where a and b are the constants. a>0, b>0 and **f(n)=O(n^k)** where k≥0. a = Number of subproblems and b = The cost of dividing and merging the subproblems.

To solve this type of recurrence relation there are 3 cases :

- **Case : 1**
  – If a = 1 then **T(n) = O(n^(k+1))** or simply **T(n) = O(n*f(n)).**
- **Case : 2**
  – If a>1 then **T(n) = O(a^(n/b) * f(n)).**
- **Case : 3**
  – If a<1 then **T(n) = O(n^k)** or simply **T(n) = O(f(n))**.

Divide-and-Conquer

```
Algo DAC(P)
{
   If (small (P))
   {
     return S(P)
   }
   else
    {
       Divide P into smaller instances P1 ,P2 ….Pk , k>=1
       Apply DAC to each of these problem (DAC(P1 ), DAC(P2 )…DAC(Pk );
       Return combine (DAC(P1 ), DAC(P2 )…DAC(Pk );
     }
}
```

Recurrence Relation Dividing functions

```
Void Test(int n)
{
    if(n>1)
     printf(‘%d’, n);
     Test(n/2);
}
```

Recurrence Relation Dividing functions

- $$T(n) = \begin{cases} 1 & n=1 \\ T(n/2) + 1 & n>1 \end{cases}$$

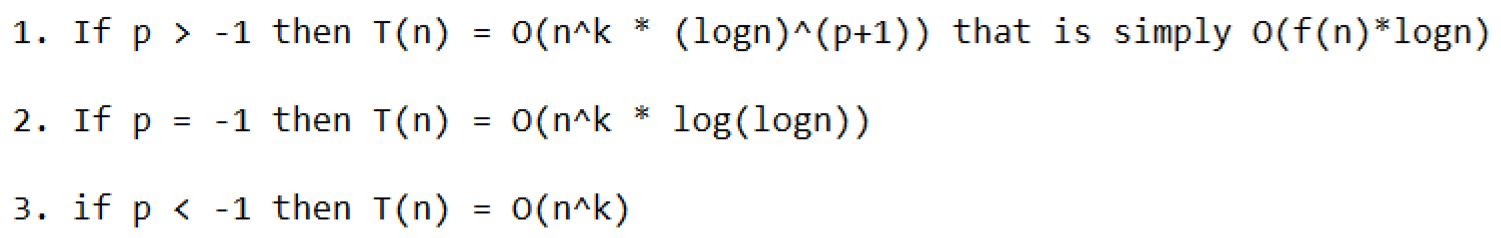
Recurrence Relation Dividing functions

```
Void Test(int n)
{
    if(n>1)
     {
      for (i=0; i<n; i++)
         {
           printf(‘%d’, n);
         }
     }
     Test(n/2);
     Test(n/2);
}
```

Recurrence Relation Dividing functions

Master theorem for Dividing functions

- The general form for the dividing functions :
- **T(n) = aT(n/b) + f(n)**, where a and b are constants. **a≥1, b>1** and **f(n) can be expressed as O(n^k * (logn)^p).** a = Number of subproblems and b = The cost of dividing and merging the subproblems.
- To find the time complexity for these kinds of functions again there are 3 cases. Here we have to find two things 1. loga base b and 2. k
- For simplicity, let's take **loga base b as x** for all below cases.
- **Case : 1**
  - If k < x then **T(n) = O(n^x).**
- **Case : 2**
  - If k = x then again some cases here comes p in the picture :



```
1. If p > -1 then T(n) = O(n^k * (logn)^(p+1)) that is simply O(f(n)*logn)

2. If p = -1 then T(n) = O(n^k * log(logn))

3. if p < -1 then T(n) = O(n^k)
```

- **Case : 3**
  - If k > x then T(n) = O(f(n)).

Recurrence Relation Dividing functions

1. Consider the recurrence T (n) = 4 T (n/2) + n
2. Consider the recurrenceT (n) = 2 T (n/2) + n log n
3. consider the recurrenceT (n) = T (n/3) + n
4. Consider the recurrence T (n) = 9 T (n/3) + n ^2.5

Example 2.11.1: Consider the recurrence T (n) = 4 T (n/2) + n

In this case, $n^{\log_b a} = n^{\log_2 4} = n^2$. Thus, we are in case 1, for f (n) is O ($n^{2-\varepsilon}$) for ε = 1. This means that T (n) is Θ (n^2) by the master.

Example 2.11.2: Consider the recurrenceT (n) = 2 T (n/2) + n log n

In this case, $n^{\log_b a} = n^{\log_2 2}$ = n. Thus, we are in case 2, with k=1, for f (n) is Θ (n log n). This means that T (n) is Θ (n $\log^2$n) by the master method.

Example 2.11.3: consider the recurrenceT (n) = T (n/3) + n

In this case $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$. Thus, we are in Case 3, for f (n) is $\Omega(n^{0+\varepsilon})$, for ε = 1, and af(n/b) = n/3 = (1/3) f(n). This means that T (n) is Θ (n) by the master method.

Example 2.11.4: Consider the recurrence T (n) = 9 T (n/3) + $n^{2.5}$

In this case, $n^{\log_b a} = n^{\log_3 9} = n^2$. Thus, we are in Case 3, since f (n) is $\Omega(n^{2+\varepsilon})$ (for ε=1/2) and af(n/b) = 9 $(n/3)^{2.5}$ = $(1/3)^{1/2}$ f(n). This means that T (n) is $\Theta(n^{2.5})$ by the master method.

Analysis of selection sort.

```
Algorithm: Selection-Sort (A)
fori ← 1 to n-1 do
   min j ← i;
   min x ← A[i]
   for j ←i + 1 to n do
      if A[j] < min x then
         min j ← j
         min x ← A[j]
   A[min j] ← A [i]
   A[i] ← min x
```

Example

Unsorted list:

5	2	1	4	3

1st iteration:

Smallest = 5

2 < 5, smallest = 2

1 < 2, smallest = 1

4 > 1, smallest = 1

3 > 1, smallest = 1

Swap 5 and 1

1	2	5	4	3

2nd iteration:

Smallest = 2

2 < 5, smallest = 2

2 < 4, smallest = 2

2 < 3, smallest = 2

No Swap

1	2	5	4	3

3rd iteration:

Smallest = 5

4 < 5, smallest = 4

3 < 4, smallest = 3

Swap 5 and 3

1	2	3	4	5

4th iteration:

Smallest = 4

4 < 5, smallest = 4

No Swap

1	2	3	4	5

Finally,

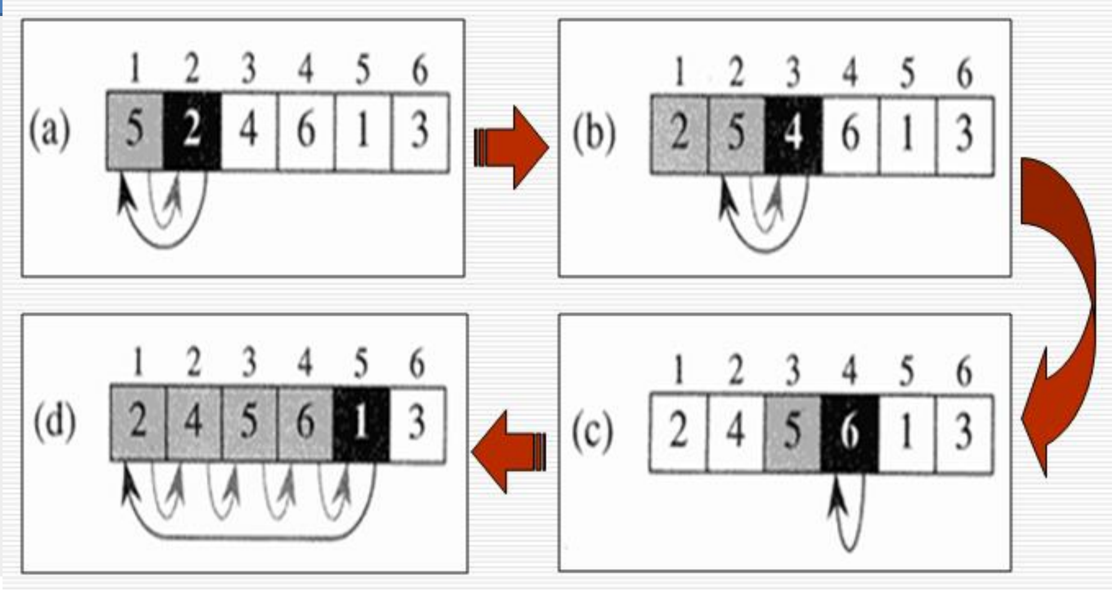
the sorted list is

1	2	3	4	5

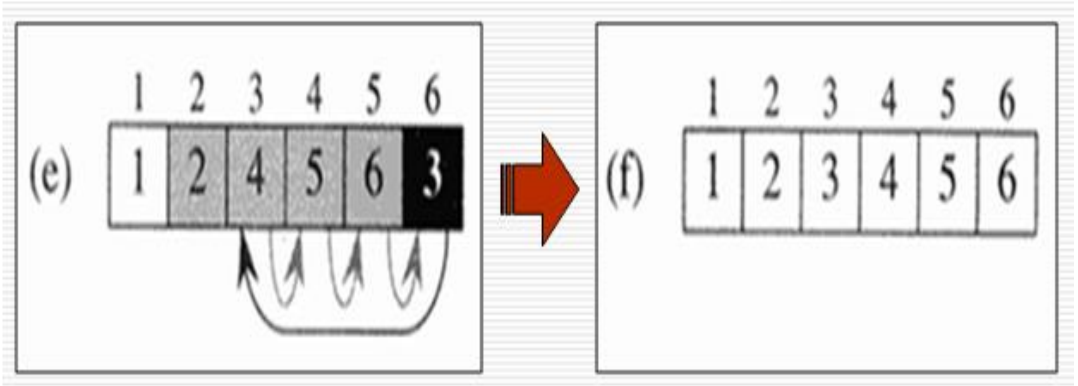
For each ***i*** from ***1*** to ***n - 1***, there is one exchange and ***n - i*** comparisons, so there is a total of ***n - 1*** exchanges and ***(n – 1) + (n – 2) + ...+ 2 + 1 =*** ***n(n – 1)/2*** comparisons.

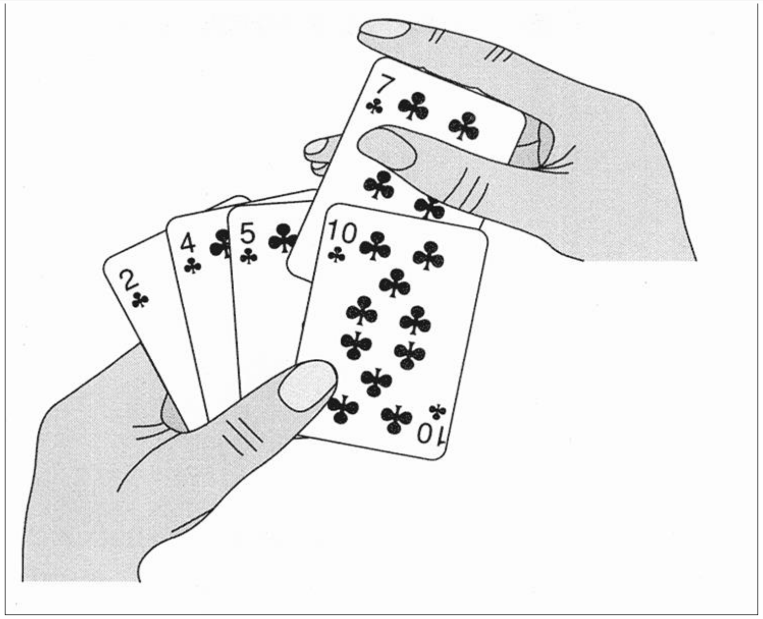
Analysis of insertion sort.

```
Algorithm: Insertion-Sort(A)
for j = 2 to A.length
   key = A[j]
   i = j - 1
   while i > 0 and A[i] > key
      A[i + 1] = A[i]
      i = i -1
   A[i + 1] = key
```



(a) 1 2 3 4 5 6 | 5 2 4 6 1 3
(b) 1 2 3 4 5 6 | 2 5 4 6 1 3
(c) 1 2 3 4 5 6 | 2 4 5 6 1 3
(d) 1 2 3 4 5 6 | 2 4 5 6 1 3





Analysis
Run time of this algorithm is very much dependent on the given input. If the given numbers are sorted, this algorithm runs in ***O(n)*** time. If the given numbers are in reverse order, the algorithm runs in ***O(n²)*** time.

Types of Complexity Classes P, NP, NP hard and NP complete

- In computer science, there exist some problems whose solutions are not yet found, the problems are divided into classes known as **Complexity Classes**. In complexity theory, a Complexity Class is a set of problems with related complexity. These classes help scientists to groups problems based on how much time and space they require to solve problems and verify the solutions. It is the branch of the theory of computation that deals with the resources required to solve a problem.
- Complexity classes are useful in organizing similar types of problems.
- **Types of Complexity Classes**
- This article discusses the following complexity classes:

1. **P Class**
2. **NP Class**
3. **NP hard**
4. **NP complete**

Types of Complexity Classes P, NP, NP hard and NP complete

<u>P Class</u>

- The P in the P class stands for **Polynomial Time.** It is the collection of decision problems(problems with a “yes” or “no” answer) that can be solved by a deterministic machine in polynomial time.
- **Features:**

1. The solution to P problems is easy to find.
2. P is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

- **This class contains many natural problems like:**
  - **Calculating the greatest common divisor.**
  - **Finding a maximum matching.**
  - **Decision versions of linear programming.**

Types of Complexity Classes P, NP, NP hard and NP complete

NP Class

- The NP in NP class stands for **Non-deterministic Polynomial Time**. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.
- **Features:**

1. The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
2. Problems of NP can be verified by a Turing machine in polynomial time.

- **Example:**
- Let us consider an example to better understand the NP class. Suppose there is a company having a total of 1000 employees having unique employee IDs. Assume that there are 200 rooms available for them. A selection of 200 employees must be paired together, but the CEO of the company has the data of some employees who can't work in the same room due to some personal reasons. This is an example of an NP problem. Since it is easy to check if the given choice of 200 employees proposed by a coworker is satisfactory or not i.e. no pair taken from the coworker list appears on the list given by the CEO. But generating such a list from scratch seems to be so hard as to be completely impractical.
- It indicates that if someone can provide us with the solution to the problem, we can find the correct and incorrect pair in polynomial time. Thus for the NP class problem, the answer is possible, which can be calculated in polynomial time.
- **This class contains many problems that one would like to be able to solve effectively:**
  - **Boolean Satisfiability Problem (SAT).**
  - **Hamiltonian Path Problem.**
  - **Graph coloring.**

Types of Complexity Classes P, NP, NP hard and NP complete

NP-hard class

- An NP-hard problem is at least as hard as the hardest problem in NP and it is the class of the problems such that every problem in NP reduces to NP-hard.
- **Features:**

1. All NP-hard problems are not in NP.
2. It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.
3. A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.

- **Some of the examples of problems in Np-hard are:**
  - **Halting problem.**
  - **Qualified Boolean formulas.**
  - **No Hamiltonian cycle.**

Types of Complexity Classes P, NP, NP hard and NP complete

Complexity Class	Characteristic feature
P	Easily solvable in polynomial time.
NP	Yes, answers can be checked in polynomial time.
Co-NP	No, answers can be checked in polynomial time.
NP-hard	All NP-hard problems are not in NP and it takes a long time to check them.
NP-complete	A problem that is NP and NP-hard is NP-complete.

Thanks