```
    }
}

string path = "";
solve(m, n, ans, srcx, srcy, visited, path);
sort(ans.begin(), ans.end());
return ans;
}
```

lec 41

Time & S. Complexity of Recursive functions

Linear Search

$$T(n) = O(n)$$

Binary Search

$$T(n) = O(\log n)$$

---

mergeSort

Steps
1) break 2 array L & R
2) Recursion → sort both
3) new array → merge L & R
4) Copy new array to orignal.

$$T(n) = K_1 + K_2 + T(n/2) + T(n/2)$$

(sort L, sort R)

$$K_3 n + K_4 n$$

(merge) (copy)

$$T(n) = K + 2T\left(\frac{n}{2}\right) + nK_5$$

$$T(n) = T\left(\frac{n}{2}\right) + nK_5$$

→ solve

$$T(n) = T\left(\frac{n}{2}\right) + nK$$
$$T(n/2) = T\left(\frac{n}{4}\right) + (n/2)K$$
$$T(n/4) = T(n/8) + (n/4)K$$
$$\vdots$$
$$T(1) = K$$

'a' times.

$a = \log n$

$$T(n) = a * nK$$
$$T(n) = \log(n) * n = O(n \log n)$$

fibonacci

$T(n) = O(2^n)$

```
f(n) {
    return f(n-1) + f(n-2)
}
```

Space Complexity

factorial $S(n) = O(n)$

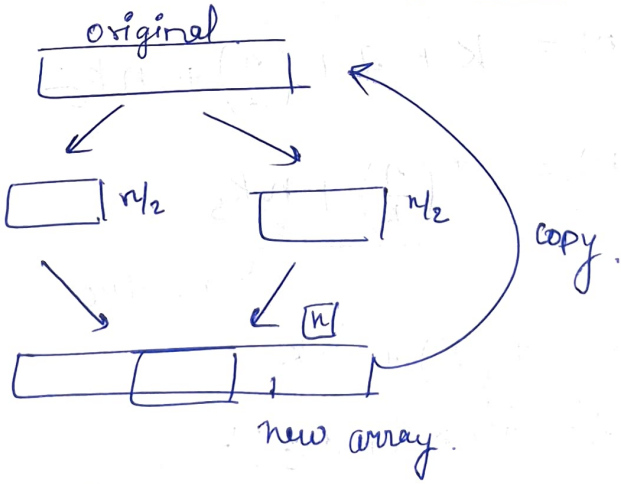
Binary Search

$S(n) = O(\log n)$

---

mergeSort ***

$S(n) = K \log n + n$



$\log n << n$

$S(n) = O(n)$ //

LEC 42

OOPS

Object oriented programming (Procedure)

obj → Real entity → state
→ behaviour

class : User Defined Datatype

Syntax

```
#include<iostream>
using
class Hero {
    char name[100];
    int health;
    char level;
};
int main() {
    Hero Aaditya; → static allocation
    return 0;
}
```

Hero Aaditya is a user defined variable aka object

Access Modifiers

↳ public, private, protected (default)

NOTE you can access properties by objectname.property

e.g. Aaditya.marks = 100;
(dot operator)

Getter & Setter

→ If properties are private then you can access those properties by creating get and set function in public section of class.

```
class Hero {
    int health;
    char level;
public:
int gethealth() {
    return health;
}
void sethealth (int h) {
    health = h;
}
};
int main() {
    Hero Aadi;
    Aadi.sethealth(96);
    cout << "Health is :" <<
        Aadi.gethealth();
    return 0;
}
```

O/P → 96

---

dynamic creation of object

```
Hero *amit = new Hero;
(*amit).gethealth();
amit -> gethealth();
```

OR — both are same

To create or destroy any obj in main we need to call constructor and distructor respectively.

when you create class, a default constructor gets created..

```
class Hero {
    private:

    public:
    Hero() {
        cout << "Called Const";
    }
};
```

Parameterised Constructor

```
public:
Hero(int health) {
    this->health = health;
}
```

(this->health: object health; health: parameter)

when data-member (property) and parameter of has same name then 'this' keyword is used.

↓ pointer (points current object).

e.g. this->health

↳ helps to access address of current object.

Copy Constructor

```
int main() {
    Hero Aadi(100, 'A');   // Parameterised constructor
    Hero Hardik(Aadi);
    return 0;
}
```

↓

default Copy constructor is called

if you want to create your own copy const then:

using pass by reference

```
Public:
Hero(Hero &temp) {
    this->health = temp->health;
    this->level = temp->level;
};
```

(temp->level: dot operator.)

* Default copy constructor does shallow copy

↳ it uses same memory from two different names.

e.g. if any data-member value of Aadi obj. is changed then it also gets changed for hardik and vice-versa.

↳ because they share same memory block using two diffrent names (aadi & hardik)

Deep copy

```
Public :
  Hero (Hero& temp) {
    char *ch = new char [100];
    strcpy (ch, temp.name);
    this → name = ch;
    this → health = health;
    this → level = level;
  }
```

Copy Assignment Operator.

```
Hero a (10,'C');
Hero b (20, 'D');

a = b;
```

now 'a' gets value of 20 & D

Destructor

↳ there is a default destructor.

~Hero()

↓

tilda (destructor starts with ~)

statically called objects calls destructor automatically.

```
{ main
  delete Aadi;
  return 0;
}
```

dynamically called obj requires manually destruction ⇒ delete ch;

Static Keyword (belongs to class not object).

we can use data member with creating any object.

1) first initial static members.
2) datatype &

```
datatype ClassName :: fieldName = Value;
int Hero :: time = 5;
```

Static functions.

↳ no need to create object to access function.

↳ static function can only access static data-members only. (doesn't have this keyword)

Lec 43

① Encapsulation (info hidding)

→ wrapping up Data Members & functions.

NOTE : Full encapsulated class :
↳ all Data Members are private.

Adv i) Security
ii) Code Reusability
iii) Unit Testing
iv) if we want, we can make class - "Read Only".

Implementation – sab private

```
class student {
  private:
    string name;
    int age;
  public:
    int getter () { return this → age; }
};
```

Inheritance

Parent class —(Data-mem) properties→ sub class / child class

```
#include

class Human {
    public:
    int height
    int age
};

class Mall : public Human {
    public:
    string name;
};
```

Base class	Public (child)	Private (child)	Protected (child)
Public	Public	Pri	Pro
Protected	Pro	Pri	Pro
Private	Not	Not accessible	Not

(Parent class)

Private data-member of any class can't be inherited.

Note: Protected data-members only works in ~~class~~ main/parent class & child class through inheritance (works like Pvt).

Types of Inheritance

1) Single level
2) Multi level
3) Multiple
4) Hybrid
5) Hierarchy.

---

① Single level

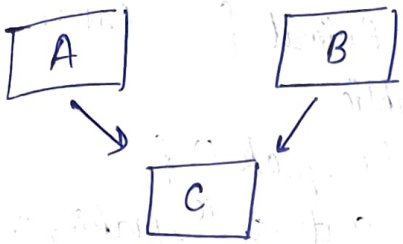
```
class Animal {
public: int age;
};

class dog : public Animal {
}
```

---

② Multi Level

```
class Animal {
    public:
        int age;
        string name;
};

class dog : public Animal
{};

class pug : public dog {
};
```

③ Multiple Inheritance.



Inheritance from more than 1 class.

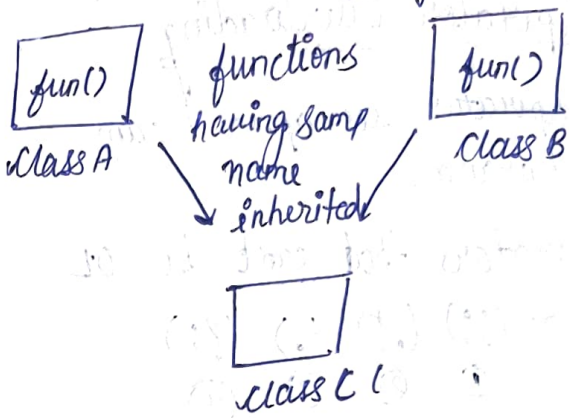
```
class teacher {
    int IQ;
};
class parent {
    int IQ2;
}
class kid : public teacher,
            public parent {

}
```

obj of class kid can access IQ and IQ2 both.

④ Hybrid Inheritance

Combination of more than 2 types of inheritance.

Inheritance Ambiguity.

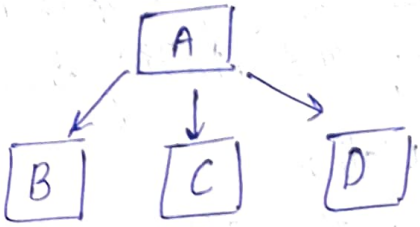



```
C obj;
obj.fun()  ✗
obj.A::fun();  ✓
obj.B::fun();  ✓
```

↓ scope resolution operator.

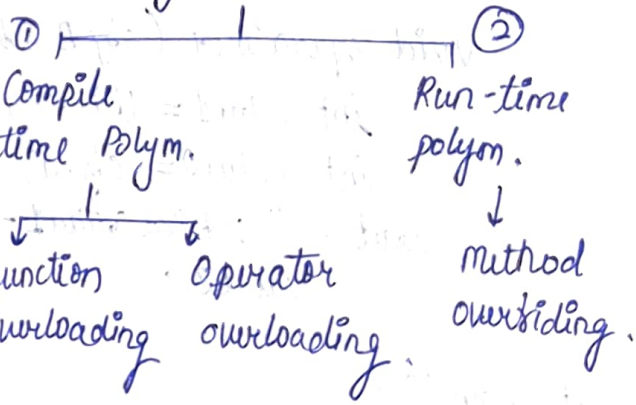
⑤ Hierarchical Type.

class behaves as parent class for more than one class



Polymorphism :

↳ existing in multiple forms.



① aka static Pdym.

② aka dynamic Polym.

Function Overloading (SAME CLASS)

→ Same Name
→ different parameters (arguments) (must)
→ different return type only X
→ different Para & return type ✓

Operator Overloading

operator se apna kaam karana.

Operators that can't be OL

→ (::) (.*) (.) (?:)
① ② ③ ④

Syntax

return_type operator+ ( ) { i/p argument.
apna kaam
}

e.g
```
#include
using
class B {
  public:
    int a;
    int b;
    void operator+ (obj B &obj){
        int value1 = this -> a;
        int value2 = obj.a;
        cout << " out " << value1 -
            value2;
    }
int main () {
    B obj1, obj2;
    obj1.a = 4;
    obj2.a = 7;
    obj1 + obj2;
```

Method Overriding / f^n overri...

function name & parameter must be same (INHERITANCE)

e.g
```
class Animal {
  public:
  void speak () {
      cout << " speaking ";
  }
};
class dog : public Animal {
  public:
  void speak () {
      cout << " barking ";
  }
};
int main () {
  dog rocky;
  rocky.speak();
  return 0;
}
```

o/p barking

Abstraction

→ Implimentation hiding
→ means displaying only essential info and hiding the detail

Encapsulation
→ Information hiding

File handling

Exception handling

→ TRY
→ CATCH
→ THROW.

(ignore)

Eg for function Overriding. using virtual function.

```
#include <iostream>
using namespace std;

class Animal {
    public:
        virtual void speak() {
            cout << "speaking" << endl;
        }
};

class Dog : public Animal {
    public:
        void speak() override {
            cout << barking << endl;
        }
};

int main() {
    Animal a;
    Dog d;
    a.speak();
    d.speak();

    Animal *ptr = &d;
    ptr -> speak();
    ↳ This is runtime polymorphism.

    return 0;
}
```

O/P speaking
barking
barking (owriding)

Rules (owrriding fn)

① Virtual function in base class
② Inhuritance is done
③ function in the derived class must have same name, return type and Parameters.
④ Owrride keyword is not compulsory but it's recommended.

Inline functions

↳ Compiler tries to expand the code in place at the point where function is called.
↳ Best suited for small & frequently called functions.
↳ Can't contain loops, recursion static variables & complex logic.

e.g syntax

```
inline int max( int a, int b){
    return (a>b)? a:b;
}
```

Friend function (similar to getter & setter)

↳ A friend func is not a class member but still can access private & protected members of class.

---

e.g

```
#include...
using...

Class Box{
    private:
    int length;

    public:
    friend void printLength (Box, b);
};
```

```
void printLength (Box b){
    cout << b.length << endl;
}
```

↳ private member

```
int main (){
    Box b;
    printLength(b);
    return 0;
}
```