



**public** **class** SumOfNumbers {

**static** **int** *result* = 0;

**public** **static** **int** sumUsingIterative(**int** n) {

**for** (**int** i = 1; i <= n; i++) {

*result* += i;

}

**return** *result*;

}

**public** **static** **int** sumUsingRecursion(**int** n) {

**if** (n == 1) {

**return** 1;

}

**return** n + *sumUsingRecursion*(n - 1);

}

}

**public** **class** AppTest {

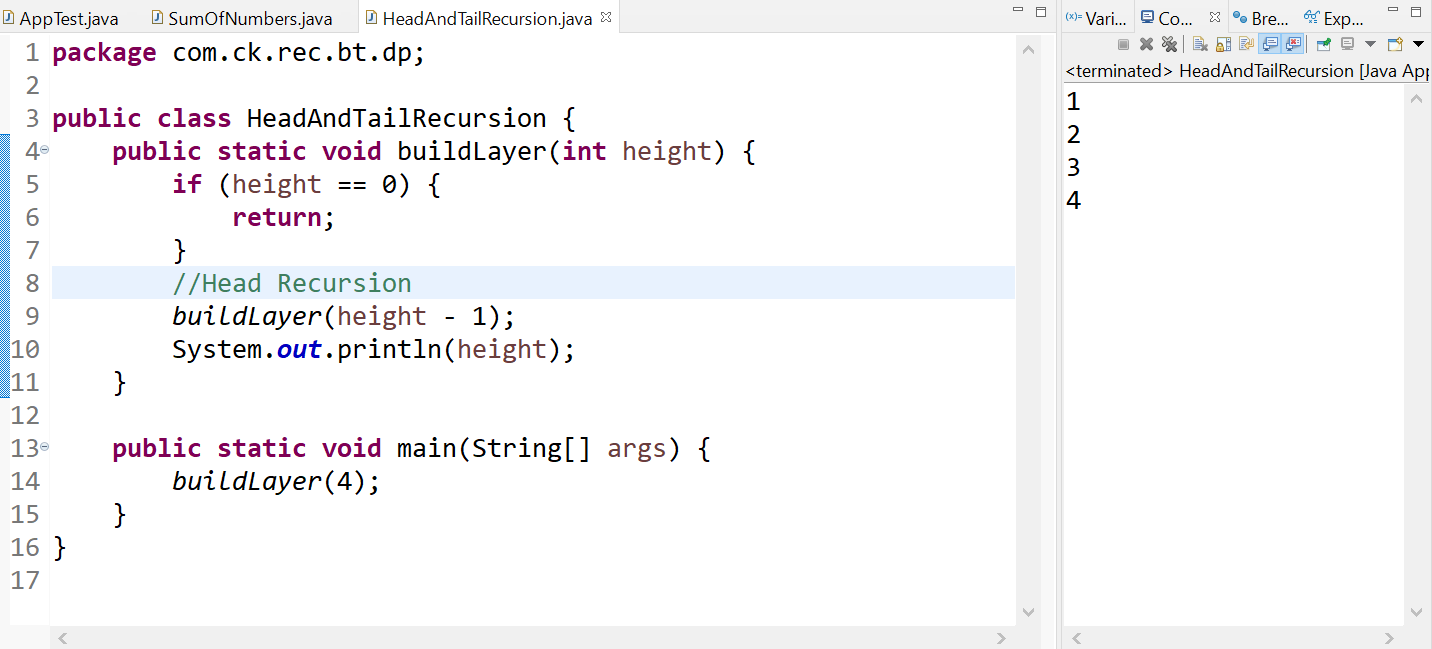
**public** **static** **void** main(String[] args) {

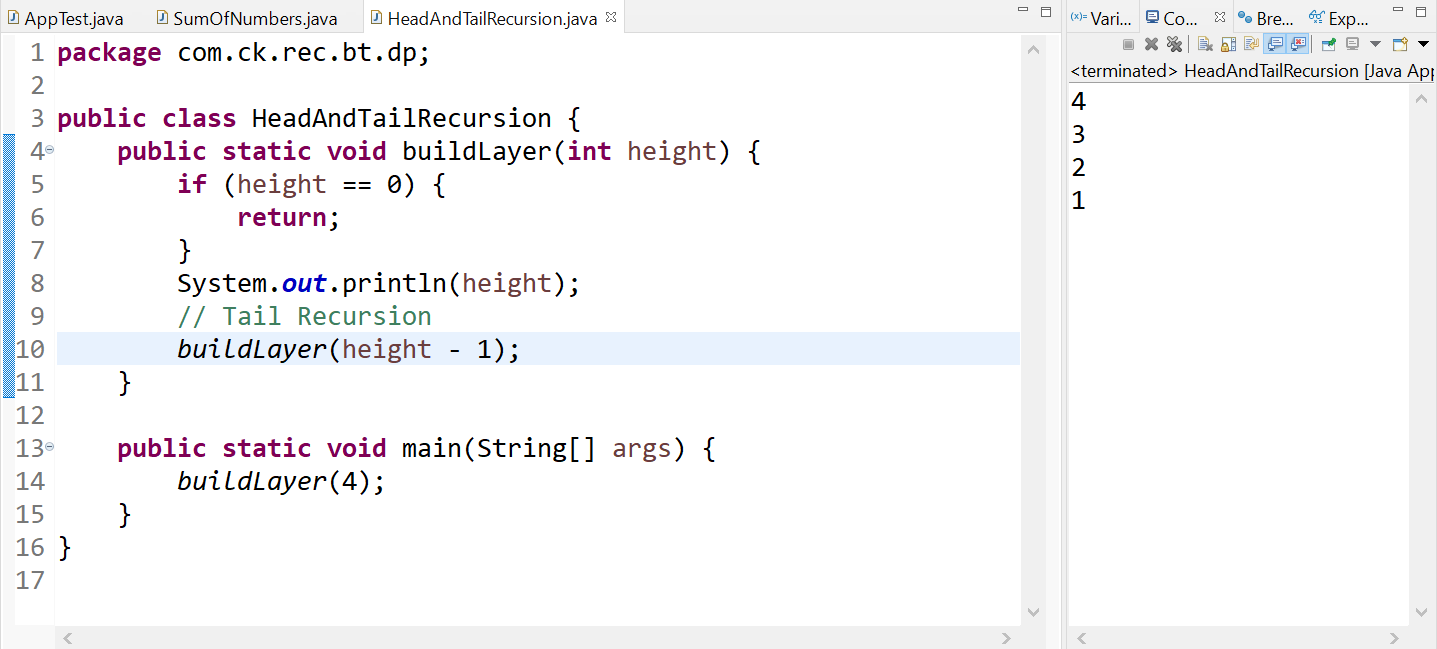
System.***out***.println(SumOfNumbers.*sumUsingIterative*(5));

System.***out***.println(SumOfNumbers.*sumUsingRecursion*(5));

}

}





* Tail Recursion is very similar to Iteration (similar to **while** or **for** loop)
* If Head recursion, we need to use OS stack to pile up the method calls and arguments.

**public** **class** Factorial {

**public** **static** **int** factorial(**int** n) {

**if** (n == 1) {

**return** 1;

}

**return** n \* *factorial*(n - 1);

}

**public** **static** **void** main(String[] args) {

System.***out***.println(*factorial*(5));

}

}

For the above problem, as discussed previously, there are two steps

1. Pushing into stack
2. Popping the result and performing multiplication.

If we see, step two is just returning the result. This can be optimized by using **accumulator**.

**public** **class** FactorialOptimized {

**public** **static** **int** factorial(**int** accumulator, **int** n) {

**if** (n == 1) {

**return** accumulator;

}

**return** *factorial*(accumulator \* n, n - 1);

}

**public** **static** **int** calculateFactorial(**int** n) {

**return** *factorial*(1, n);

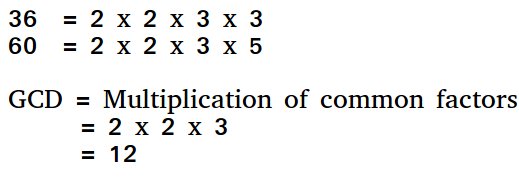
}

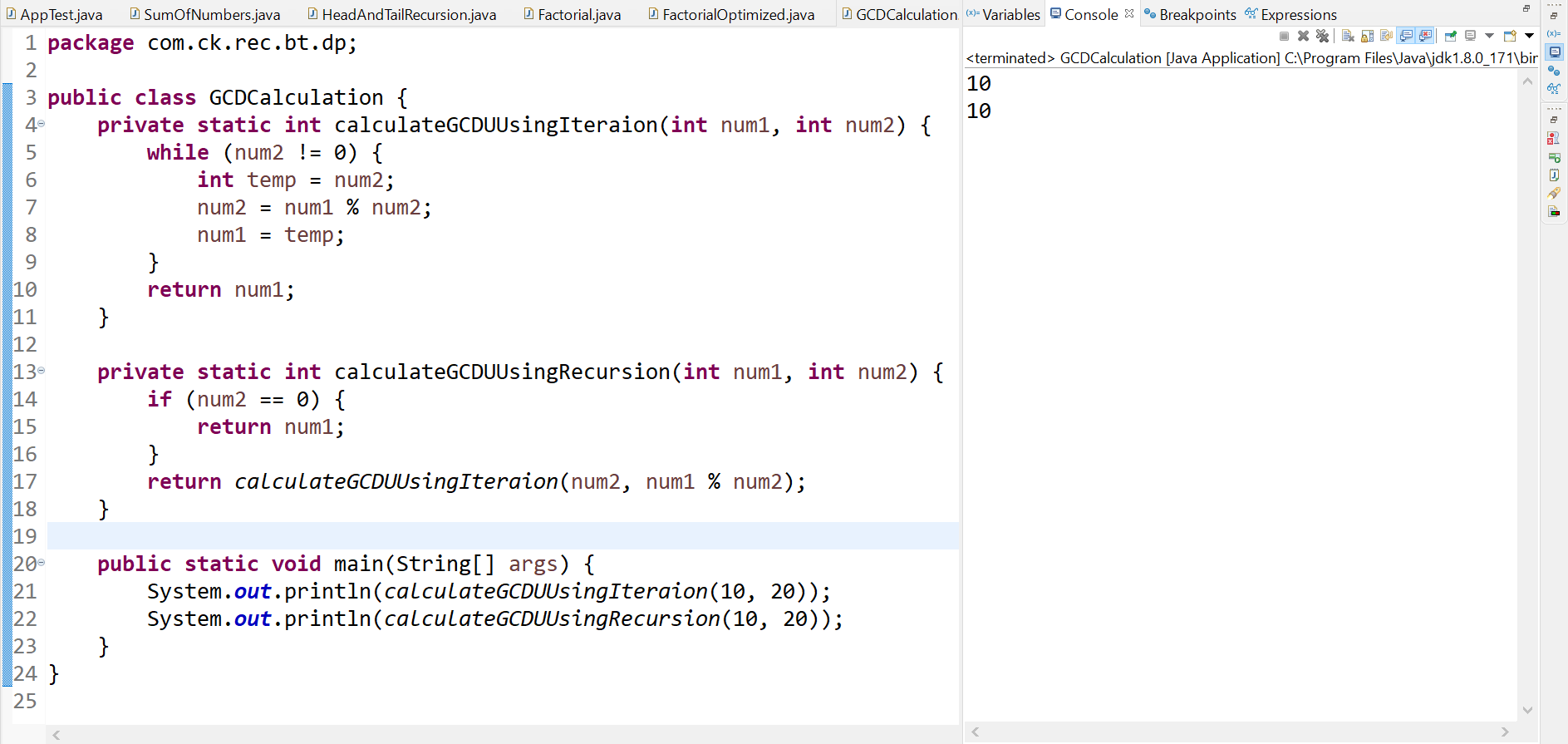
**public** **static** **void** main(String[] args) {

System.***out***.println(*calculateFactorial*(5));

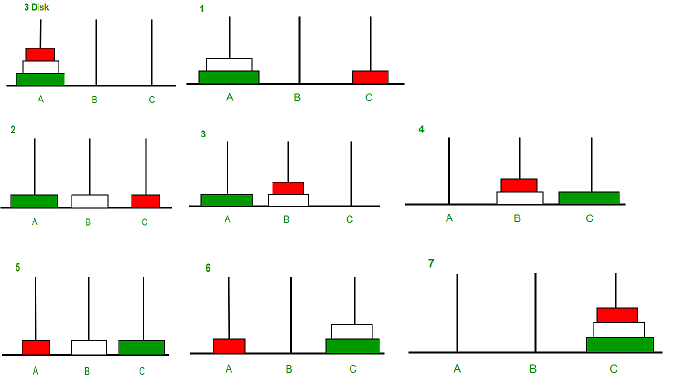
}

}





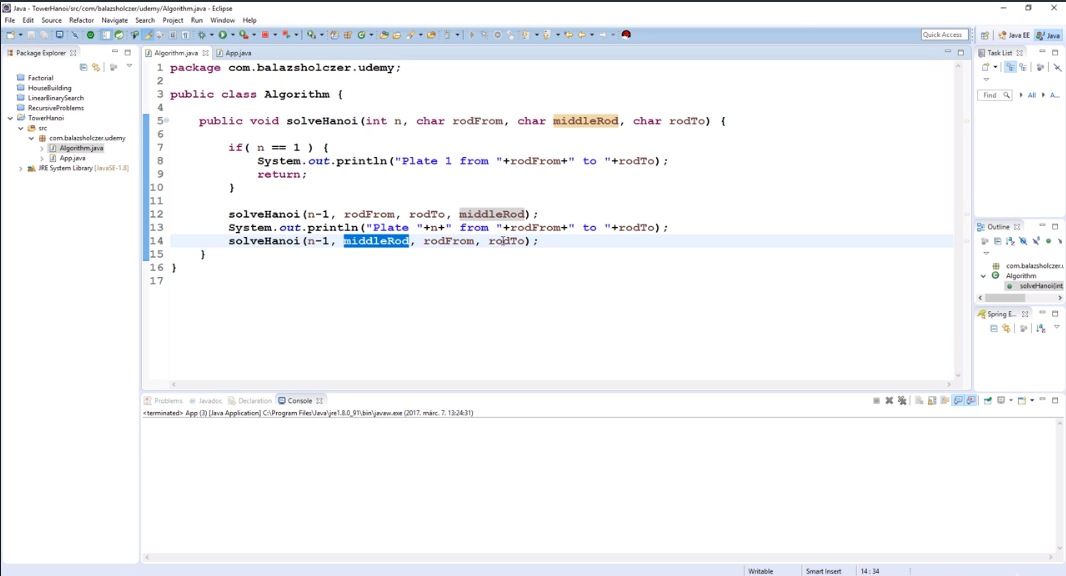
**Towers of Hanoi**

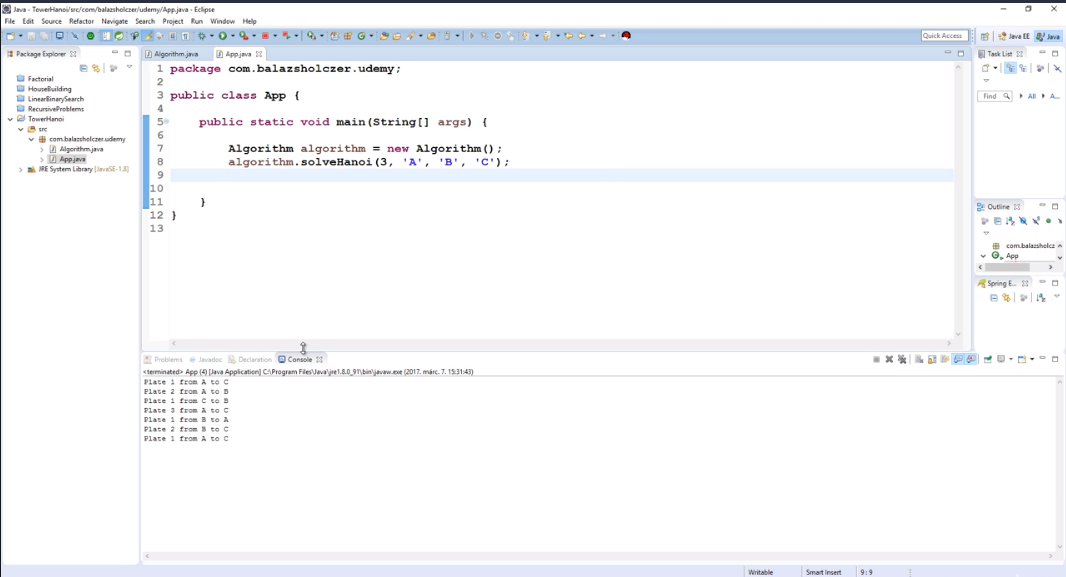


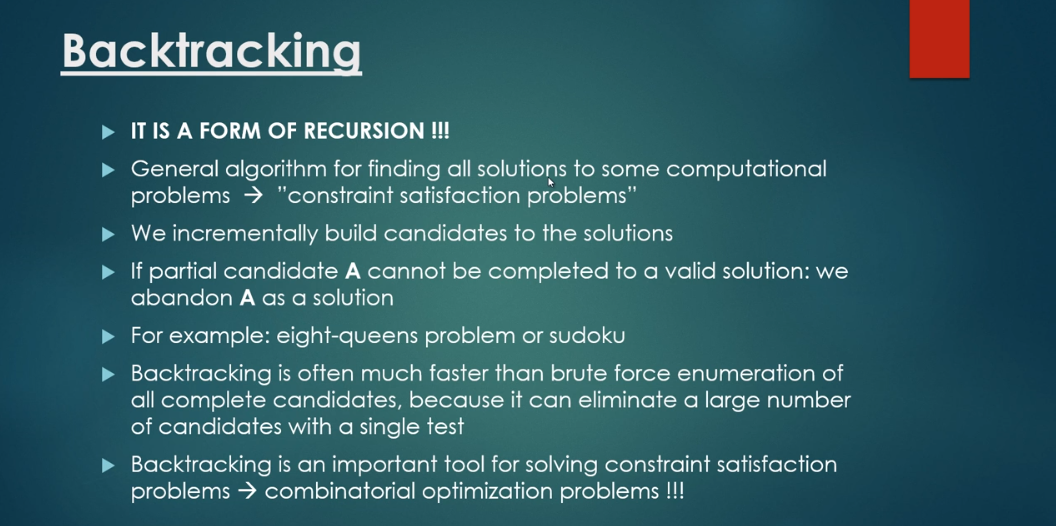
If you see, on step 4, this becomes the sub problem to step 1.

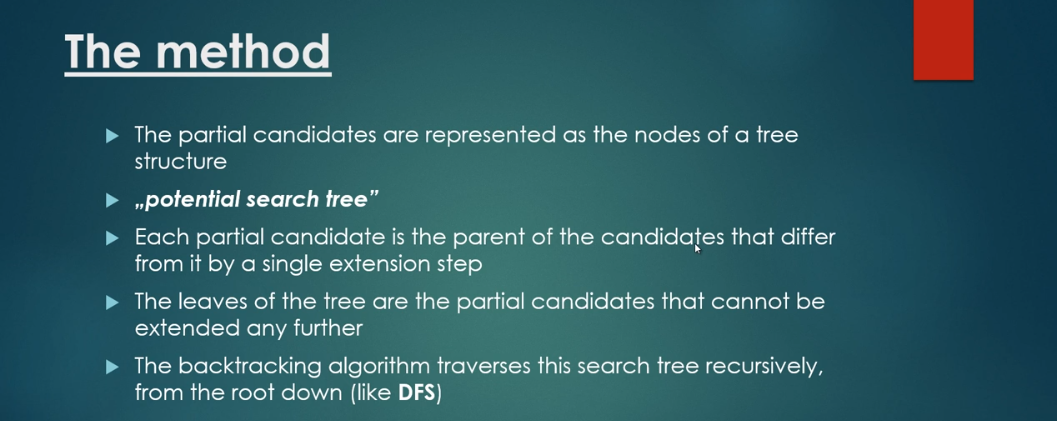
But here, A becomes auxiliary rod. In step 1, the auxiliary rod is rod c. This gives us the idea that we can solve this problem with recursion as the sub problem can be solved after a few steps.

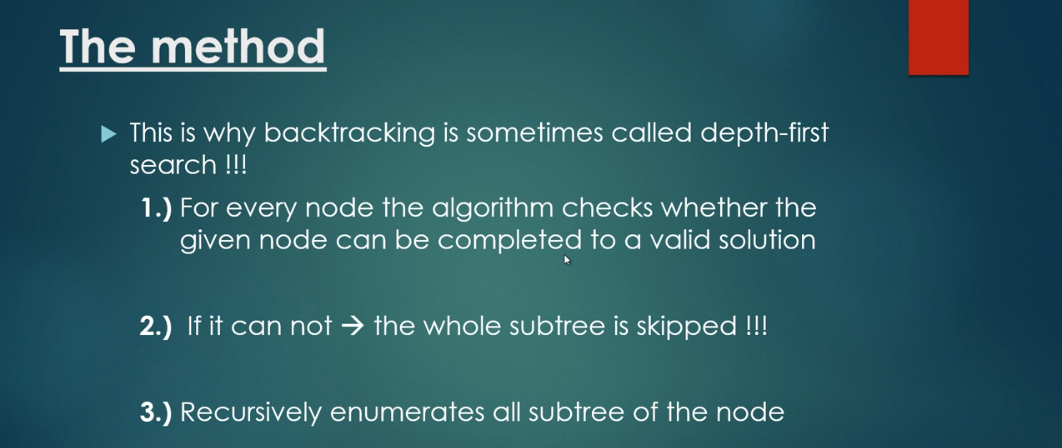
It takes (**2^n) -1** steps to solve the above problem. Hence Time complexity is **O(2^n)**

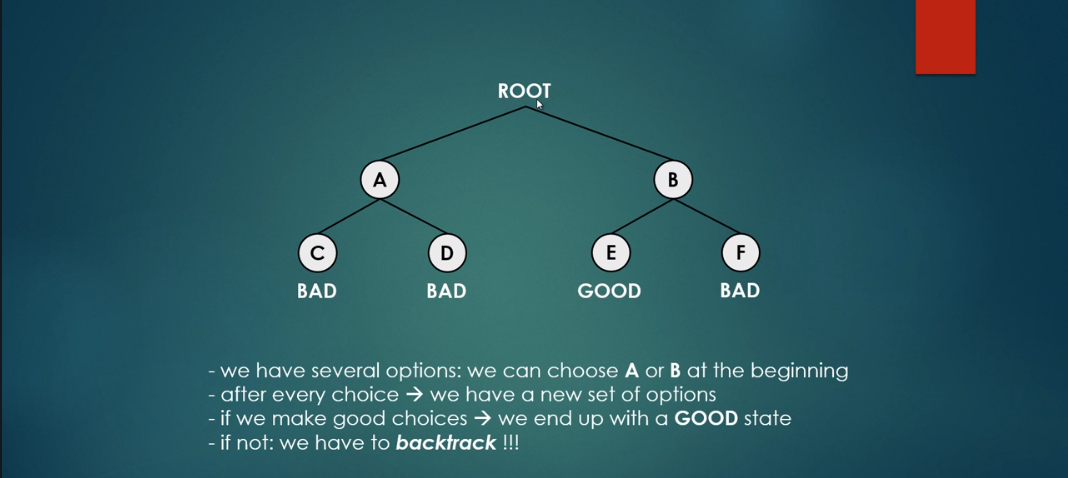


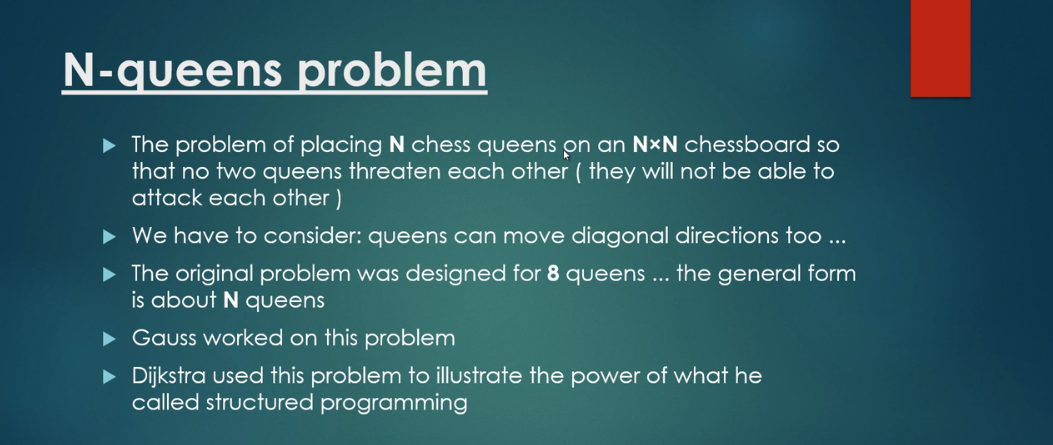












**We can discard the nodes like in red, as we know that those cases are not needed to be considered. This saves us from considering a lot of cases as show below.**

