Chapter 3

Recursion

3.1 Introduction

3.2 class IntStack

```java
 * File Name: IntStack.java int stack of Infinite capacity
/*
 * To compile you require: IntUtil.java RandomInt.java IntArray.java IntStack.java
 */

class IntStack {
  /*
   * Private
   */
  private int sp;
  private int max; // max element seen by stack in its whole life time
  private IntArray stack;
  private boolean display = false;
  private static final IntUtil u = new IntUtil();

  public void setDisplay(boolean x) {
    display = x;
    stack.setDisplay(x);
  }

  public IntStack() {
    sp = 0;
    max = 0;
    stack = new IntArray();
    stack.setDisplay(display);
  }

  public boolean isEmpty() {
    return (sp == 0) ? true : false;
  }

  public boolean isFull() {
    return false;
  }

  public int size() {
    return sp;
  }

  public int maxStackSize() {
    return max;
  }

  public void push(int v) {
    stack.set(sp, v);
    if (max < sp) {
      max = sp; // /max element seen by stack in its whole life time
    }
    sp++;
  }

  public int pop() {
    if (isEmpty()) {
      System.out.println("Stack is empty " + sp);
      u.myassert(false);
    }
    return (stack.get(--sp));
```

```java
}

public int top() {
  if (isEmpty()) {
    System.out.println("Stack is empty " + sp);
    u.myassert(false);
  }
  return stack.get(sp - 1);
}

/*
 * fromtop = true: get from sp-1 to 0 fromtop = false: get from 0 to sp-1
 */
public int[] toArray(boolean fromtop) {
  if (isEmpty()) {
    return null;
  }
  if (fromtop == false) {
    int[] a = stack.toarray(0, sp - 1);
    return a;
  } else {
    int[] a = stack.toarray(sp - 1, 0);
    return a;
  }
}

/*
 * Print from zero to sp-1
 */
public void pFromZeroToSp() {
  if (isEmpty() == false) {
    stack.p(0, sp - 1);
  }
}

public void pLnFromZeroToSp() {
  pFromZeroToSp();
  System.out.println();
}

/*
 * print from sp-1 to 0
 */
public void p() {
  if (isEmpty() == false) {
    stack.p(sp - 1, 0);
  }
}

public void p(String t) {
  System.out.print(t);
  p();
}

public void pLn() {
  p();
  System.out.println();
```

```java
  }

  public void pLn(String t) {
    System.out.print(t);
    pLn();
  }

  /*
   * Test routines
   */
  private static void test1() {
    IntStack s = new IntStack();
    for (int i = 0; i < 8; ++i) {
      s.push(i);
    }
    s.pLn("After Pushing 8 elements: ");
    s.pop();
    s.pop();
    s.pop();
    s.pLn("After Poping 3 elements: ");
    s.push(-90);
    s.push(-80);
    s.pLn("After Pushing -90 and -80: ");
    System.out.println("Poping off until empty");
    while (s.isEmpty() == false) {
      System.out.print(s.top() + " ");
      s.pop();
    }
    System.out.println();
    System.out.println("Max stack size: " + s.maxStackSize());
  }

  private static void testBench() {
    System.out.println("------------test1--------------------");
    test1();
    System.out.println("------------DONE!--------");
  }

  public static void main(String[] args) {
    System.out.println("IntStack.java");
    testBench();
    System.out.println("Done");
  }
}
```

3.3 Printing stars

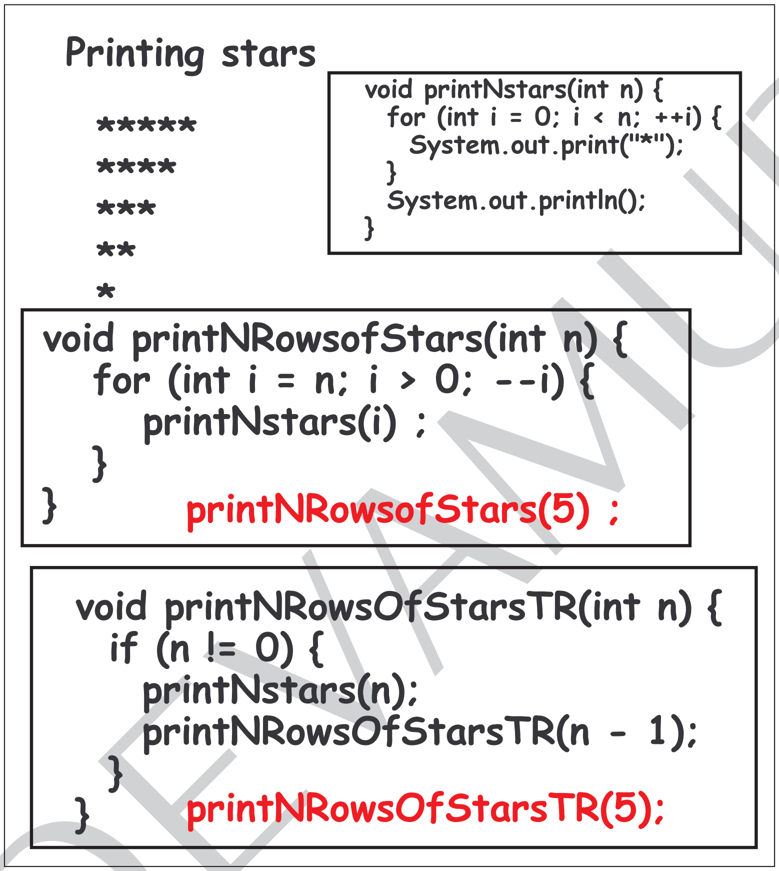


Figure 3.1: Printing starts

3.4 Computing factorial of a number

3.4.1 Computing factorial of a number using recursion

Factorial of a number n

```
!5 = 5 * 4 * 3 * 2 * 1
!n = n * n-1 * n-2 *    * 2 * 1
```

```
int fact(int n) {
  int ans = 1;
  for (int i = n; i >= 2; --i) {
    ans = ans * i;
  }
  return ans;
}
```

```
!5 = 5 * 4 * 3 * 2 * 1
!4 = 4 * 3 * 2 * 1

!5 = 5 * !4
```

```
int factR(int n) {
  if (n == 0) {
    return 1;
  }
  return n * factR(n - 1);
}
```

Stack overflow for n = 5000 ;

```
int factS(int n) {
  int t = n;
  IntStack s = new IntStack();
  while (n > 0) {
    s.push(n--);
  }
  int ans = 1;
  while (!(s.isEmpty())) {
    ans = ans * s.top();
    s.pop();
  }
  return ans;
}
```

Figure 3.2: Computing factorial using recursion

3.4.2 Complexity of computing factorial of a number using recursion

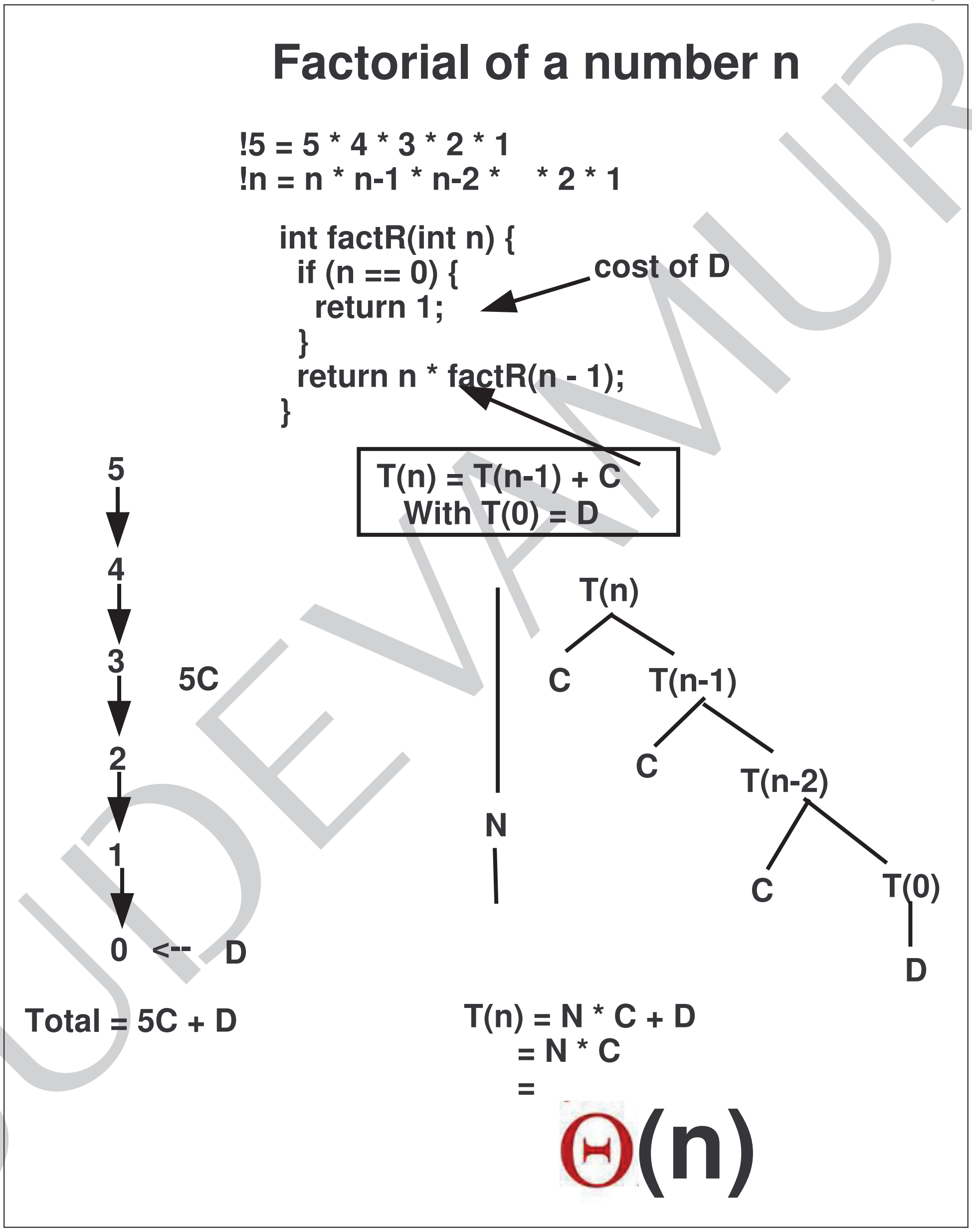


Figure 3.3: Computing T(n)

3.4.3 Computing factorial of a number using tail recursion

Factorial of a number n using Tail recursion

!5 = 5 * 4 * 3 * 2 * 1
!n = n * n-1 * n-2 * * 2 * 1

```
int factTR(int n, int ans) {
  if (n == 0) {
    return ans;
  }
  return factTR(n - 1, ans*n);
}

int factWithTailR(int n) {
  return factTR(n, 1);
}

int r = factWithTailR(5) ;
```

```
int factTRS(int n) {
  IntStack s = new IntStack();
  int ans = 1;
  s.push(ans);
  while (n > 0) {
    ans = s.pop() * n--;
    s.push(ans);
  }
 return ans;
}

int ans = factTRS(50000);
```

STACK SIZE OF 1

NO Stack overflow

Figure 3.4: Computing factorial using tail recursion

3.5 Computing sum of a number

3.5.1 Computing sum of a number using recursion

Sum of a number n

S(5) = 5 + 4 + 3 + 2 + 1 + 0
S(n) = n + n-1 + n-2 + + 2 + 1 + 0

```
int sum(int n) {
  int ans = 0;
  for (int i = n; i >= 1; --i) {
    ans = ans + i;
  }
  return ans;
}
```

S(5) = 5 + 4 + 3 + 2 + 1 + 0
S(4) = 4 + 3 + 2 + 1 + 0

S(5) = 5 + S(4)

```
int sumR(int n) {
  if (n == 0) {
    return 0;
  }
  return n + sumR(n - 1);
}
```

Stack overflow for n = 5000 ;

```
int sumS(int n) {
  IntStack s = new IntStack();
  while (n > 0) {
    s.push(n--);
  }
  int ans = 0;
  while (!(s.isEmpty())) {
    ans = ans + s.top();
    s.pop();
  }
  return ans;
}
```

Figure 3.5: Computing sum using recursion

3.5.2 Complexity of computing sum of a number using recursion

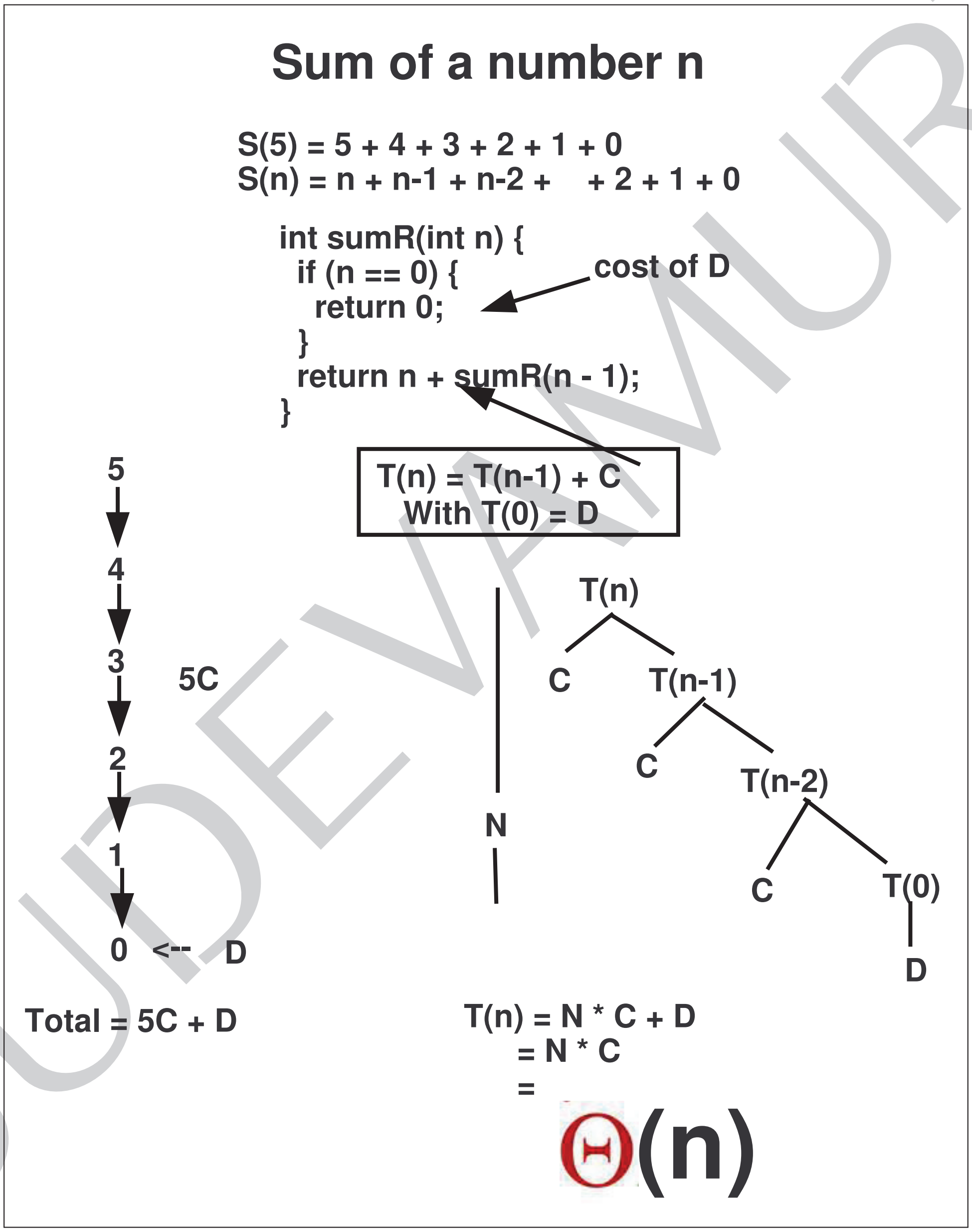


Figure 3.6: Computing T(n)

3.5.3 Computing sum of a number using tail recursion

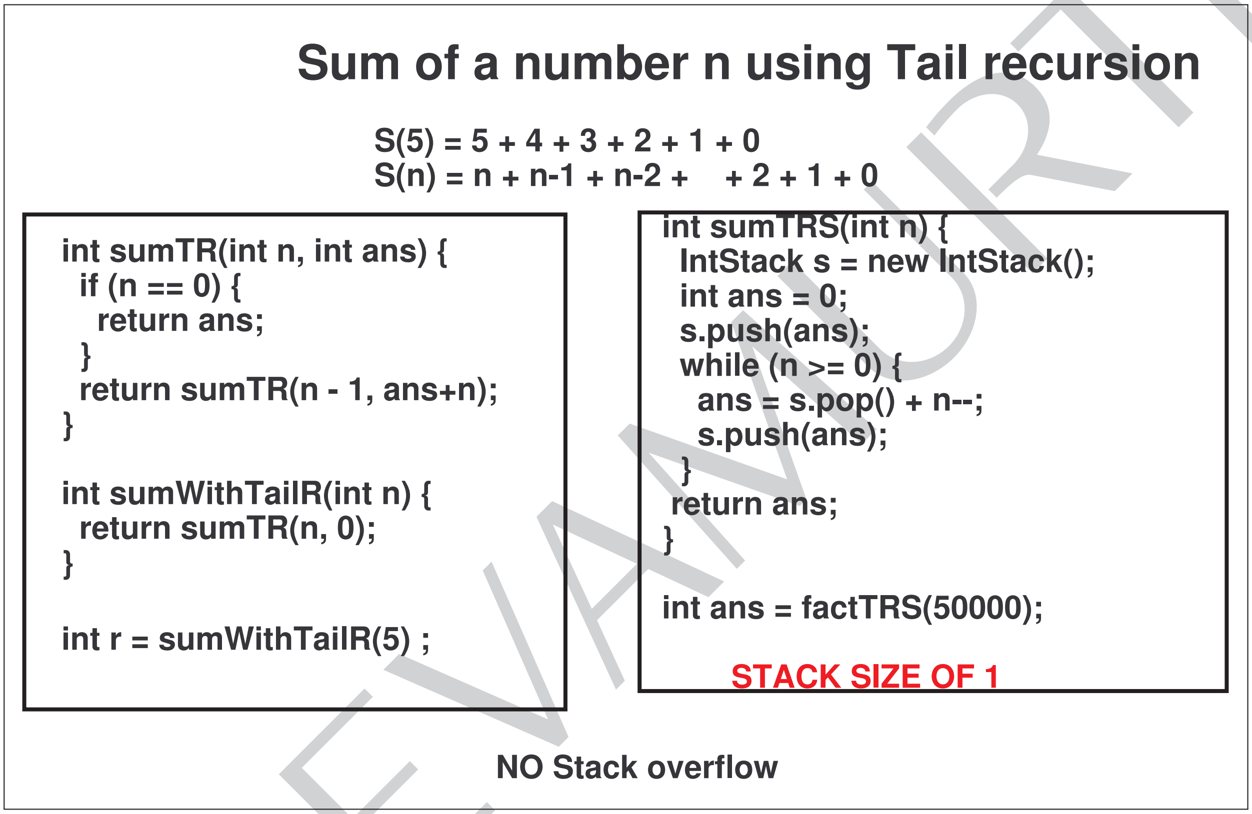


Figure 3.7: Computing sum using tail recursion

3.6 Computing number of binary digits to represent a decimal number

3.6.1 Computing number of binary digits to represent a decimal number using recursion

Number of Binary Digit to Represent a Decimal number N

D	B
0	0
1	1
2	01
3	11
4	100
5	101
15	1111
16	10000

```
int numBinaryDigit(int n) {
  int ans = 1; //n = 0 and 1 requires 1 digit
  if (n > 1) {
    int t = n;
    ans = 0;
    while (t > 0) { //9
      ++ans;
      t = t / 2; //10
    }
  }
  return ans;
}
```

```
2)15(7
  2)7(3
    2)3(1
      2)1(0
```

```
int numBinaryDigitS(int n) {
  IntStack s = new IntStack();
  while (n > 0) {
    n = n / 2;
    s.push(n);
  }
  int ans = 0;
  while (!(s.isEmpty())) {
    ans = ans + 1;
    s.pop();
  }
  return ans;
}
```

```
int numBinaryDigitR_r(int n) {
  if (n > 1) {
    return (1 + numBinaryDigitR_r(n / 2));
  }
  return 1; //n = 0 and 1 requires 1 digit
}
```

Figure 3.8: Computing number of binary digit using recursion

3.6.2 Complexity of computing number of binary digits to represent a decimal number

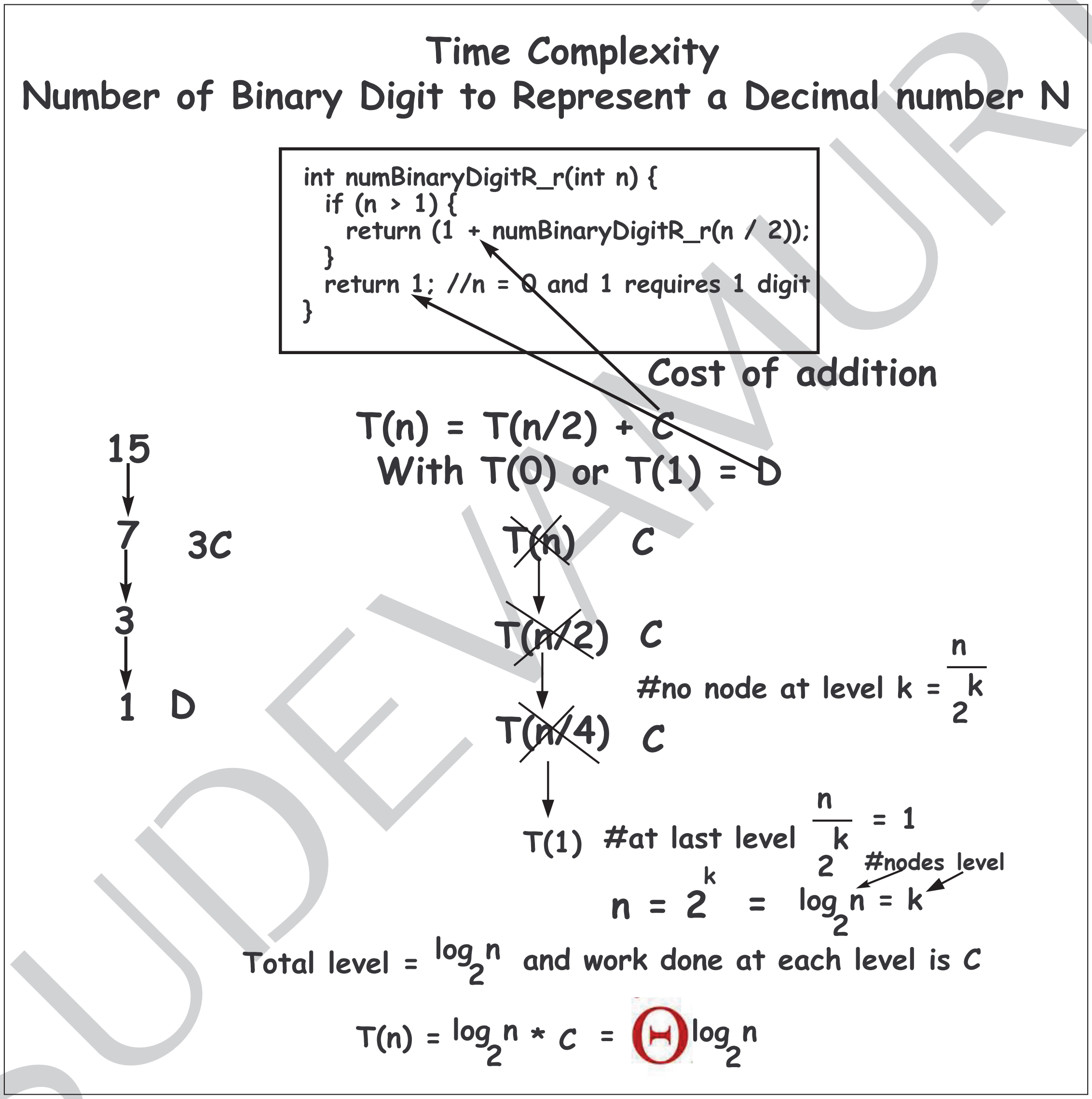


Figure 3.9: Computing T(n)

3.6.3 Computing number of binary digits to represent a decimal number using tail recursion

```
private static int numBinaryDigitTR_r(int n, int ans) {
    incKount() ;
    if (n > 1) {
      return numBinaryDigitTR_r(n/2,ans+1);
    }
    return ans ; //n = 0 and 1 requires 1 digit
}

private static int numBinaryDigitTR(int n) {
    setKount() ;
    return numBinaryDigitTR_r(n,1);
}
```

```
private static int numBinaryDigitTRS(int n) {
    setKount() ;
    IntStack s = new IntStack();
    int t = n ;
    int ans = 1 ;
    s.push(ans) ;
    while (n > 1) {
      ans = ans + 1 ;
      s.pop();
      n = n/2;
      s.push(n) ;
    }
    setKount(s.maxStackSize());
    System.out.println("numBinaryDigitTRS " + t +
        ": = " + ans + " max Stack used = " + getKount()) ;
    return ans ;
}
```

```
numBinaryDigitTRS 0: = 1 max Stack used = 1
numBinaryDigitTRS 1: = 1 max Stack used = 1
numBinaryDigitTRS 10: = 4 max Stack used = 1
numBinaryDigitTRS 15: = 4 max Stack used = 1
numBinaryDigitTRS 16: = 5 max Stack used = 1
numBinaryDigitTRS 65535: = 16 max Stack used = 1
numBinaryDigitTRS 65536: = 17 max Stack used = 1
```

Figure 3.10: Computing T(n)

3.7 Fibonacci number

3.7.1 Fibonacci number using iteration

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}	F_{20}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

```
private static int fib(int n) {
    setKount(0) ;
    int prevprev = 0 ;
    int prev = 1 ;
    int ans = 0 ;
    for (int i = 0 ; i <= n ; ++i) {
      incKount() ;
      if (i <= 1) {
        ans = i ;
      }else {
        ans = prev + prevprev ;
        prevprev = prev ;
        prev = ans ;
      }
    }
    System.out.println("Fib of " + n + ": = " + ans + " num executed = " + getKount()) ;
    return ans ;
  }
```

$\Theta(n)$

Constant space

```
for (int i = 0; i < 300; ++i) {//Will Never hang
   int a = fib(i) ;
}
```

```
Fib of 0: = 0 num executed = 1
Fib of 1: = 1 num executed = 2
Fib of 2: = 1 num executed = 3
Fib of 3: = 2 num executed = 4
Fib of 4: = 3 num executed = 5
Fib of 5: = 5 num executed = 6
Fib of 6: = 8 num executed = 7
Fib of 7: = 13 num executed = 8
Fib of 8: = 21 num executed = 9
Fib of 9: = 34 num executed = 10
Fib of 10: = 55 num executed = 11
Fib of 11: = 89 num executed = 12
Fib of 12: = 144 num executed = 13
Fib of 13: = 233 num executed = 14
```

```
Fib of 287: = -1084667583 num executed = 288
Fib of 288: = 38092160 num executed = 289
Fib of 289: = -1046575423 num executed = 290
Fib of 290: = -1008483263 num executed = 291
Fib of 291: = -2055058686 num executed = 292
Fib of 292: = 1231425347 num executed = 293
Fib of 293: = -823633339 num executed = 294
Fib of 294: = 407792008 num executed = 295
Fib of 295: = -415841331 num executed = 296
Fib of 296: = -8049323 num executed = 297
Fib of 297: = -423890654 num executed = 298
Fib of 298: = -431939977 num executed = 299
Fib of 299: = -855830631 num executed = 300
```

Figure 3.11: Fibonacci number using iteration

3.7.2 Fibonacci number using recursion

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}	F_{20}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

```
private static int fib_r1(int n) {
    incKount() ;
    if (n <= 1) {
      return n ;
    }
    return (fib_r1(n-2) + fib_r1(n-1)) ;
  }
```

T(n) = 1 for n = 0/1
     = T(n-2)+T(n-2) +C

$$\Theta(2^n)$$

```
private static int fib_r(int n) {
    setKount(0) ;
    int a = fib_r1(n) ;
    System.out.println("Fib of " + n + ": = " + a + " num executed = " + getKount()) ;
    return a ;
  }
```

```
for (int i = 0; i < 40; ++i) { //Run with 300 to see hangs at 50
     int a = fib_r(i) ;
  }
```

```
Fib of 0: = 0 num executed = 1
Fib of 1: = 1 num executed = 1
Fib of 2: = 1 num executed = 3
Fib of 3: = 2 num executed = 5
Fib of 4: = 3 num executed = 9
Fib of 5: = 5 num executed = 15
Fib of 6: = 8 num executed = 25
Fib of 7: = 13 num executed = 41
Fib of 8: = 21 num executed = 67
Fib of 9: = 34 num executed = 109
Fib of 10: = 55 num executed = 177
Fib of 11: = 89 num executed = 287
Fib of 12: = 144 num executed = 465
Fib of 13: = 233 num executed = 753
Fib of 14: = 377 num executed = 1219
Fib of 15: = 610 num executed = 1973
Fib of 16: = 987 num executed = 3193
Fib of 17: = 1597 num executed = 5167
Fib of 18: = 2584 num executed = 8361
Fib of 19: = 4181 num executed = 13529
Fib of 20: = 6765 num executed = 21891
Fib of 21: = 10946 num executed = 35421
Fib of 22: = 17711 num executed = 57313
Fib of 23: = 28657 num executed = 92735
Fib of 24: = 46368 num executed = 150049
Fib of 25: = 75025 num executed = 242785
Fib of 26: = 121393 num executed = 392835
Fib of 27: = 196418 num executed = 635621
Fib of 28: = 317811 num executed = 1028457
Fib of 29: = 514229 num executed = 1664079
Fib of 30: = 832040 num executed = 2692537
Fib of 31: = 1346269 num executed = 4356617
Fib of 32: = 2178309 num executed = 7049155
Fib of 33: = 3524578 num executed = 11405773
Fib of 34: = 5702887 num executed = 18454929
Fib of 35: = 9227465 num executed = 29860703
Fib of 36: = 14930352 num executed = 48315633
Fib of 37: = 24157817 num executed = 78176337
Fib of 38: = 39088169 num executed = 126491971
Fib of 39: = 63245986 num executed = 204668309
```

Figure 3.12: Fibonacci number using recursion

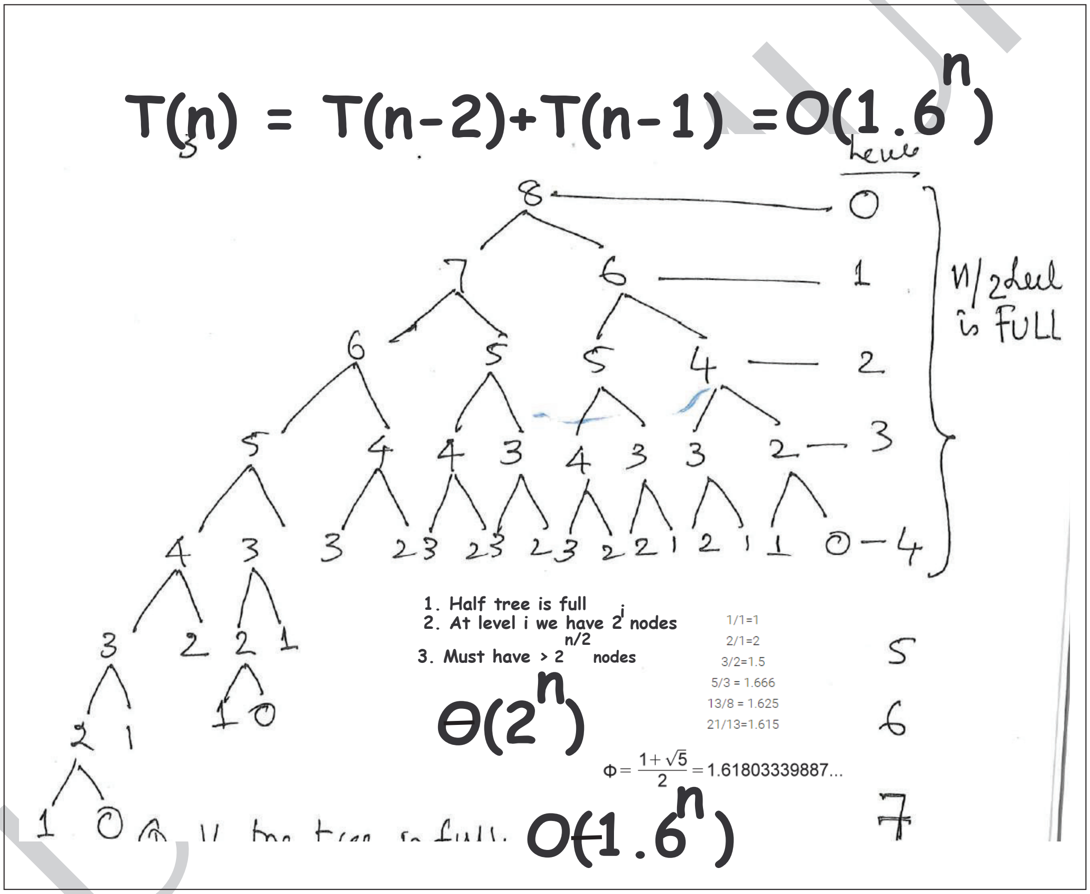


Figure 3.13: Complexity of computing Fibonacci number using recursion

3.7.3 Fibonacci number using tail recursion

```
private static int fibTR(int n, int prevprev, int prev) {
    incKount() ;
    if (n == 0) {
      return prevprev ;
    }
    if (n == 1) {
      return prev ;
    }
    return fibTR(n-1,prev,prev+prevprev);
  }
```

```
private static int fibWithTailR(int n) {
    setKount() ;
    int a = fibTR(n,0,1);
    System.out.println("fibWithTailR of " + n + ": = " + a + " num executed = " + getKount()) ;
    return a ;
  }
```

```
for (int i = 0; i < 300; ++i) { //Will Never hang
   int a = fibWithTailR(i) ;
 }
```

```
fibWithTailR of 0: = 0 num executed = 1
fibWithTailR of 1: = 1 num executed = 1
fibWithTailR of 2: = 1 num executed = 2
fibWithTailR of 3: = 2 num executed = 3
fibWithTailR of 4: = 3 num executed = 4
fibWithTailR of 5: = 5 num executed = 5
fibWithTailR of 6: = 8 num executed = 6
fibWithTailR of 7: = 13 num executed = 7
fibWithTailR of 8: = 21 num executed = 8
fibWithTailR of 9: = 34 num executed = 9
fibWithTailR of 10: = 55 num executed = 10
fibWithTailR of 11: = 89 num executed = 11
fibWithTailR of 12: = 144 num executed = 12
fibWithTailR of 13: = 233 num executed = 13
fibWithTailR of 14: = 377 num executed = 14
```

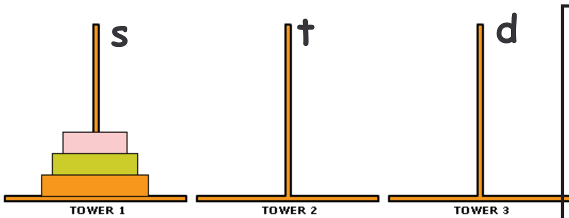
```
fibWithTailR of 290: = -1008483263 num executed = 290
fibWithTailR of 291: = -2055058686 num executed = 291
fibWithTailR of 292: = 1231425347 num executed = 292
fibWithTailR of 293: = -823633339 num executed = 293
fibWithTailR of 294: = 407792008 num executed = 294
fibWithTailR of 295: = -415841331 num executed = 295
fibWithTailR of 296: = -8049323 num executed = 296
fibWithTailR of 297: = -423890654 num executed = 297
fibWithTailR of 298: = -431939977 num executed = 298
fibWithTailR of 299: = -855830631 num executed = 299
```

Figure 3.14: Fibonacci number using tail recursion

3.7.4 Fibonacci number using stack

3.8 Tower of Hanoi

3.8.1 Tower of Hanoi using recursion




```
void TH(int n, int s, int t, int d) {
    if (n) {
        TH(n-1,s,d,t);
         s -> d
        TH(n-1,t,s,d) ;
    }
}                    TH(3,s,t,d)
```

$$T(n) = T(n-1) + 1 + T(n-1)$$
$$= 2T(n-1) + 1$$

$$T(1) = 1$$
$$T(2) = 2(T(1))+1 = 3 = 2^2-1$$
$$T(3) = 2(T(2))+1 = 7 = 2^3-1$$
$$T(4) = 2(T(3))+1 = 15 = 2^4-1$$
$$T(5) = 2(T(4))+1 = 31 = 2^5-1$$
$$T(n) = 2(T(n-1))+1 = 2^n-1 = \Theta(2^n)$$

Figure 3.15: Tower of Hanoi using recursion and computing T(n)

3.8.2 Tower of Hanoi using stack

3.9 Printing permutation of n numbers

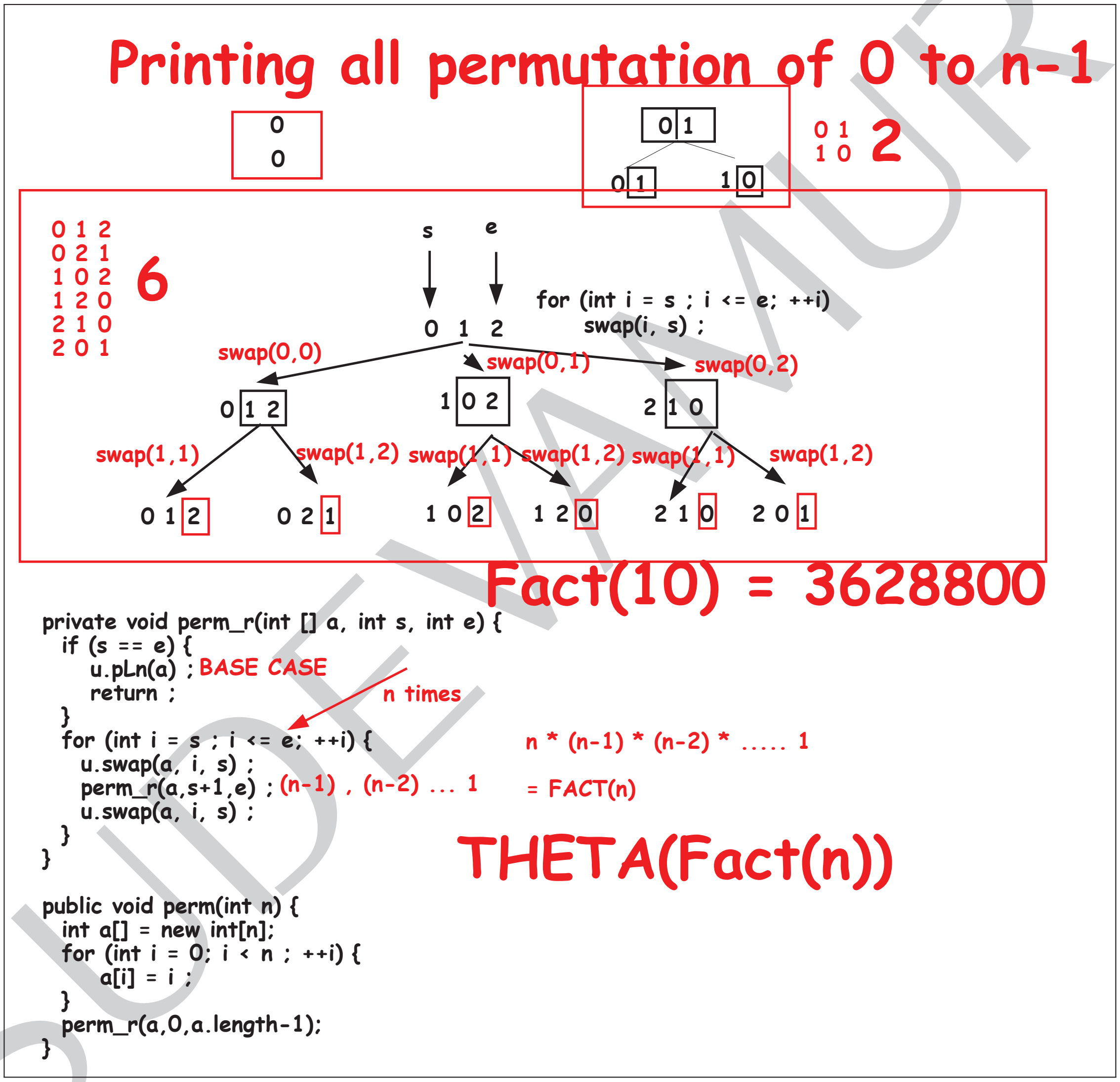


Figure 3.16: Printing all permutation

3.10 Recursion trees