Analysis of Recursive Algorithms

- What is a recurrence relation?
- Forming Recurrence Relations
- Solving Recurrence Relations
- Analysis Of Recursive Factorial method
- Analysis Of Recursive Selection Sort
- Analysis Of Recursive Binary Search
- Analysis Of Recursive Towers of Hanoi Algorithm

What is a recurrence relation?

- A recurrence relation, T(n), is a recursive function of integer variable n.
- Like all recursive functions, it has both recursive case and base case.
- Example:

$$T(n) = \begin{cases} a & \text{if } n = 1 \\ 2T(n/2) + bn + c & \text{if } n > 1 \end{cases}$$

- The portion of the definition that does not contain T is called the **base case** of the recurrence relation; the portion that contains T is called the **recurrent or recursive case**.
- Recurrence relations are useful for expressing the running times (i.e., the number of basic operations executed) of recursive algorithms

Forming Recurrence Relations

- For a given recursive method, the base case and the recursive case of its recurrence relation correspond directly to the base case and the recursive case of the method.
- Example 1: Write the recurrence relation for the following method.

```
public void f (int n) {
   if (n > 0) {
        System.out.println(n);
        f(n-1);
   }
}
```

- The base case is reached when n == 0. The method performs one comparison. Thus, the number of operations when n == 0, T(0), is some constant a.
- When n > 0, the method performs two basic operations and then calls itself, using ONE recursive call, with a parameter n – 1.
- Therefore the recurrence relation is:

$$
\begin{array}{ll}
T(0) = a & \text{for some constant a} \\
T(n) = b + T(n-1) & \text{for a constant b}
\end{array}
$$

Forming Recurrence Relations

- Example 2: Write the recurrence relation for the following method.

```
public int g(int n) {
   if (n == 1)
      return 2;
   else
      return 3 * g(n / 2) + g( n / 2) + 5;
}
```

- The base case is reached when n == 1. The method performs one comparison and one return statement. Therefore, T(1), is constant **c**.
- When **n > 1**, the method performs **TWO** recursive calls, each with the parameter n **/ 2,** and some constant # of basic operations.
- Hence, the recurrence relation is:

$$T(1) = c \qquad \text{for some constant } c$$
$$T(n) = b + 2T(n / 2) \qquad \text{for a constant } b$$

Solving Recurrence Relations

- To solve a recurrence relation T(n) we need to derive a form of T(n) that is not a recurrence relation. Such a form is called a closed form of the recurrence relation.

- There are four methods to solve recurrence relations that represent the running time of recursive methods:
  - Iteration method (*unrolling and summing)*
  - Substitution method
  - Recursion tree method
  - Master method

- In this course, we will only use the Iteration method.

Solving Recurrence Relations - Iteration method

- Steps:
  - Expand the recurrence
  - Express the expansion as a summation by plugging the recurrence back into itself until you see a pattern.
  - Evaluate the summation
- In evaluating the summation one or more of the following summation formulae may be used:
- Arithmetic series:

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

- Geometric Series:

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} \quad (x \neq 1)$$

$$\sum_{k=0}^{n-1} x^k = \frac{x^n - 1}{x - 1} \quad (x \neq 1)$$

- Special Cases of Geometric Series:

$$\sum_{k=0}^{n-1} 2^k = 2^n - 1$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \quad \text{if } x < 1$$

Solving Recurrence Relations - Iteration method

- Harmonic Series:

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \ln n$$

- Others:

$$\sum_{k=1}^{n} \lg k \approx n \lg n$$

$$\sum_{k=0}^{n-1} c = cn.$$

$$\sum_{k=0}^{n-1} \frac{1}{2^k} = 2 - \frac{1}{2^{n-1}}$$

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=0}^{n} k(k+1) = \frac{n(n+1)(n+2)}{3}$$

Analysis Of Recursive Factorial method

- Example1: Form and solve the recurrence relation for the running time of factorial method and hence determine its big-O complexity:

```
long factorial (int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial (n – 1);
}
```

T(0) = c

T(n) = b + T(n - 1)

= b + b + T(n - 2)

= b +b +b + T(n - 3)

…

= kb + T(n - k)

When k = n, we have:

T(n) = nb + T(n - n)

= bn + T(0)

= bn + c.

Therefore method factorial is O(n).

Analysis Of Recursive Selection Sort

```
public static void selectionSort(int[] x) {
   selectionSort(x, x.length - 1);}
private static void selectionSort(int[] x, int n) {
   int minPos;
   if (n > 0) {
      minPos = findMinPos(x, n);
      swap(x, minPos, n);
      selectionSort(x, n - 1);
   }
}
private static int findMinPos (int[] x, int n) {
   int k = n;
   for(int i = 0; i < n; i++)
      if(x[i] < x[k])  k = i;
   return k;
}
private static void swap(int[] x, int minPos, int n) {
   int temp=x[n]; x[n]=x[minPos]; x[minPos]=temp;
}
```

Analysis Of Recursive Selection Sort

- findMinPos is O(n), and swap is O(1), therefore the recurrence relation for the running time of the selectionSort method is:

$T(0) = a$

$T(n) = T(n-1) + n + c \qquad n > 0$

$= [T(n-2) + (n-1) + c] + n + c = T(n-2) + (n-1) + n + 2c$

$= [T(n-3) + (n-2) + c] + (n-1) + n + 2c = T(n-3) + (n-2) + (n-1) + n + 3c$

$= T(n-4) + (n-3) + (n-2) + (n-1) + n + 4c$

$= \ldots\ldots$

$= T(n-k) + (n-k+1) + (n-k+2) + \ldots\ldots + n + kc$

When k = n, we have :

$$T(n) = T(0) + 1 + 2 + \ldots\ldots + n + nc$$

$$= a + \sum_{i=0}^{n} i + cn$$

$$= a + \left(\frac{n(n+1)}{2}\right) + cn$$

$$= \frac{n^2}{2} + \left(c + \frac{1}{2}\right)n + a$$

Therefore, Recursive Selection Sort is **$O(n^2)$**

Analysis Of Recursive Binary Search

```
public int binarySearch (int target, int[] array,
                          int low, int high) {
   if (low > high)
      return -1;
   else {
      int middle = (low + high)/2;
      if (array[middle] == target)
         return middle;
      else if(array[middle] < target)
         return binarySearch(target, array, middle + 1, high);
      else
         return binarySearch(target, array, low, middle - 1);
     }
}
```

- The recurrence relation for the running time of the method is:

  $T(1) = a$ if $n = 1$ (one element array)

  $T(n) = T(n / 2) + b$ if $n > 1$

Analysis Of Recursive Binary Search

Expanding:

$$T(n) = T(n / 2) + b$$
$$= [T(n / 4) + b] + b = T(n / 2^2) + 2b$$
$$= [T(n / 8) + b] + 2b = T(n / 2^3) + 3b$$
$$= \ldots\ldots\ldots$$
$$= T(n / 2^k) + kb$$

When $n / 2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$, we have:

$$T(n) = T(1) + b \log_2 n$$
$$= a + b \log_2 n$$

Therefore, Recursive Binary Search is **O(log n)**

Analysis Of Recursive Towers of Hanoi Algorithm

```
public static void hanoi(int n, char from, char to, char temp){
   if (n == 1)
      System.out.println(from + " --------> " + to);
   else{
      hanoi(n - 1, from, temp, to);
      System.out.println(from + " --------> " + to);
      hanoi(n - 1, temp, to, from);
   }
}
```

- The recurrence relation for the running time of the method **hanoi** is:

$$T(n) = a \qquad \text{if } n = 1$$

$$T(n) = 2T(n - 1) + b \qquad \text{if } n > 1$$

Analysis Of Recursive Towers of Hanoi Algorithm

Expanding:

T(n) = 2T(n – 1) + b

= 2[2T(n – 2) + b] + b **= 2^2 T(n – 2) + 2b + b**

= 2^2 [2T(n – 3) + b] + 2b + b **= 2^3 T(n – 3) + 2^2b + 2b + b**

= 2^3 [2T(n – 4) + b] + 2^2b + 2b + b **= 2^4 T(n – 4) + 2^3 b + 2^2b + 2^1b + 2^0b**

= ……

= 2^k T(n – k) + b[2^{k-1} + 2^{k-2} + . . . 2^1 + 2^0]

$$= 2^k T(n-k) + b\sum_{i=0}^{k-1} 2^i$$

$$= 2^k T(n-k) + b(2^k - 1)$$

When k = n – 1, we have:

$$\begin{aligned} T(n) &= 2^{n-1} T(1) + b(2^{n-1} - 1) \\ &= (a+b)\, 2^{n-1} - b \\ &= \left(\frac{a+b}{2}\right) 2^n - b \end{aligned}$$

Therefore, The method **hanoi** is **O(2^n)**