King Fahd University of Petroleum & Minerals
College of Computer Science & Engineering

Information & Computer Science Department

Unit 6

Analysis of Recursive Algorithms

Information and Computer Science ICS

Reading Assignment

- This set of slides.

Information and Computer Science

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 describing the **number of comparisons** carried out 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 number of comparisons is 1, and hence, T(0) = 1.
- When n > 0,
  - The number of comparisons is 1 + T(n-1).
- Therefore the recurrence relation is:

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

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 2: Write the recurrence relation describing the **number of System.out.println** statements executed 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 number of executed System.out.println's is 0, i.e., T(0) = 0.
- When n > 0,
  - The number of executed System.out.println's is 1 + T(n-1).
- Therefore the recurrence relation is:

$$T(n) = \begin{cases} 0 & n = 0 \\ T(n-1)+1 & n > 0 \end{cases}$$

Forming Recurrence Relations

- Example 3: Write the recurrence relation describing the **number of additions** carried out 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 $n = 1$ and hence, $T(1) = 0$
- When **n > 1**, $T(n) = 2T(n/2) + 2$
- Hence, the recurrence relation is:

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T\left(\frac{n}{2}\right) + 2 & n > 1 \end{cases}$$

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:

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

•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

- Example 1: Form and solve the recurrence relation describing the **number of multiplications** carried out by the factorial method and hence determine its big-O complexity:

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

$$T(n) = \begin{cases} 0 & n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

$$= n \qquad n \geq 0$$

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 describing the number of times **the printing statement** is executed for the method **hanoi** is:

Analysis Of Recursive Towers of Hanoi Algorithm

- The recurrence relation describing the number of times the printing statement is executed for the method **hanoi** and its solution is:

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

$$= 2^n - 1 \qquad n \geq 1$$

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 describing the number of **element comparisons** for the method is:

Analysis Of Recursive Binary Search

- The recurrence relation describing the number of element comparisons for the method in the worst case and its solution are:

$$T(n) = \begin{cases} 0 & n = 0 \\ T\left(\frac{n-1}{2}\right) + 1 & n \geq 1 \text{ (assuming } n = 2^k - 1) \end{cases}$$

$$= \log_2(n+1) \qquad n \geq 0$$

Analysis Of Recursive Selection Sort

- Example 2: Form and solve the recurrence relation describing the **number of element comparisons** **(`x[i] > x[k]`)** carried out by the selection sort method and hence determine its big-O complexity:

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

Analysis Of Recursive Selection Sort

- Example 2: Form and solve the recurrence relation describing the **number of element comparisons** `(x[i] > x[k])` carried out by the selection sort method and hence determine its big-O complexity:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(n-1) + n - 1 & n > 1 \end{cases}$$

$$= \frac{(n-1)n}{2} \qquad n \geq 1$$